	**ArchWare** Architecting Evolvable Software www.architecture-ware.org	European RTD Project IST-2001-32360 

The ArchWare ADL: Definition of the Textual Concrete Syntax

(Project Deliverable D1.2b)

Authors: Sorana Cimpan, InterUnec – Univ. de Savoie,
Flavio Oquendo, InterUnec – Univ. de Savoie,
Dharini Balasubramaniam, Univ. of St. Andrews,
Graham Kirby, Univ. of St. Andrews,
Ron Morrison, Univ. of St. Andrews

Editors: Sorana Cimpan, InterUnec – Univ. de Savoie
<sorana.cimpan@univ-savoie.fr>
Flavio Oquendo, InterUnec – Univ. de Savoie
<flavio.oquendo@univ-savoie.fr>

Document ID: D1.2b

Date: 31 December 2002

Version: 1.0

Status: Issued

Reviewed by: Ferdinando Gallo – CPR,
Radu Mateescu – INRIA,
Bob Snowdon – Univ. of Manchester

Distribution: Public

Abstract: This deliverable presents the textual concrete syntax of the core and style-based ArchWare Architecture Description Language (ADL).

Acknowledgments

The design of the textual concrete syntax of the ArchWare Architecture Description Language (ADL) is the result of a team effort: Prof. Flavio Oquendo and Dr. Sorana Cimpan (Software Engineering Group of the InterUnec – University of Savoie at Annecy), Prof. Ron Morrison, Dr. Dharini Balasubramaniam and Dr. Graham Kirby (Persistent Programming Group of the University of St. Andrews).

We would like to acknowledge our colleagues in the ArchWare project, in particular the internal reviewers and the members of the other team involved in WP1/WP4: Prof. Warboys' group (Informatics Process Group of the University of Manchester). We would also like to acknowledge the other partners (CPR, INRIA, THESAME, and ENGINEERING) for their feedbacks.

Caveat

This deliverable is subject to evolutions based on feedbacks from its usage within the project. Only the most recent version of this document should be used. New releases of this document will be advertised to Project Participants and be made available on the project cooperative work support web site.

Disclaim

TABLE OF CONTENTS

DOCUMENT HISTORY AND EVOLUTIONS

Changes Log			
Document Version	**Ch. Request Origin**	**Changes and Reasons for Change**	**Authors**
1.0a – v. 1		*Initial Version*: Definition of the core concrete syntax of the ArchWare ADL.	S. Cimpan, F. Oquendo
1.0a – v..2		*Issue 1.0a*: Contributions from Univ. of St. Andrews and insights from Univ. of Manchester (Bob Snowdon, Brian Warboys).	S. Cimpan, F. Oquendo, D. Balasubramaniam, G. Kirby, R. Morrison
1.0b (public dissemination)		*Issue 1.0b*: Definition of the style-based concrete syntax of the ArchWare ADL. Revision according to insights from Univ. of St. Andrews and Univ. of Manchester.	S. Cimpan, F. Oquendo, D. Balasubramaniam, G. Kirby, R. Morrison

Open Issues/To Do Items					
Issue Number	**Date Logged**	**Description of Open Issue/ To Do Item**	**Priority**	**Status**	**Origin**

I. INTRODUCTION

I.1 Document Purpose

This deliverable presents the textual concrete syntax of the ArchWare Architecture Description Language (ADL). This concrete syntax is defined in line with the abstract syntax and formal semantics defined in deliverable D1.1b [Oquendo et al. 2002].

I.2 Document Overview

The ArchWare Architecture Description Language (ADL) is a formal language for modelling evolvable software architectures. It is part of the ArchWare Architectural Languages, which are:

- the ArchWare Architecture Description Language1 (ADL),
- the ArchWare Architecture Analysis Language[2] (AAL),
- the ArchWare Architecture Refinement Language[3] (ARL),
- the ArchWare Architecture eXchange Language[4] (AXL).

The ArchWare/ADL is defined as a layered language:

- The inner layer, called π-ADL, is the core layer providing the core behaviour constructs and the core structure constructs for modelling software architectures.
- The outer layer, built on the core layer, called σπ-ADL, provides the style constructs, from which the base component-connector style and other derived styles can be defined.

This deliverable focuses on the definition of the textual concrete syntax of the ArchWare/ADL, including the π-ADL and the σπ-ADL. This textual concrete syntax corresponds to the abstract syntax and formal semantics defined in the deliverable D1.1b (*Definition of the ArchWare/Core-ADL and Style-ADL Abstract Syntax and Formal Semantics*) [Oquendo et al. 2002].

[1] The ArchWare ADL is defined in deliverables D1.1 (abstract syntax and formal semantics), D1.2 (this one: textual concrete syntax), and D1.4 (graphical concrete syntax).

[2] The ArchWare AAL is defined in deliverable D3.1.

[3] The ArchWare ARL is defined in deliverable D6.1.

[4] The ArchWare AXL is defined in deliverable D1.3.

I.3 Document Structure

This document presents the concrete syntaxes of the outer layers of the language, the π-ADL and the σπ-ADL. The sub-layers are presented in deliverable D1.6 (*Definition of the ArchWare/Core-ADL and Style-ADL Reference Model*) together with the code generation rules.

This document is organised as follows:

- chapter II defines the concrete syntax of π-ADL: it includes context-free and context-sensitive definitions,
- chapter III defines the concrete syntax of the σπ-ADL layer: it includes context-free and context-sensitive definitions,
- chapter IV presents case studies that illustrates the use of the π-ADL and the σπ-ADL,
- appendixes A and B summarises the context free syntax and typing rules of π-ADL.

II. π-ADL: TEXTUAL CONCRETE SYNTAX

The π-ADL concrete syntax is defined in terms of context free and context sensitive definitions.

Formal syntactic rules are used to define the context free syntax of the π-ADL. The context free syntactic rules are further qualified by a set of context sensitive type rules. The formal semantics of the syntactic categories are given in the deliverable D1.1b (*Definition of the ArchWare/Core-ADL and Style-ADL Abstract Syntax and Formal Semantics*) [Oquendo et al. 2002]. The formal syntactic rules define the set of all syntactically legal π-ADL declarations, knowing that the meaning of these declarations are defined by the semantics in deliverable D1.1b.

Production Rules

To define the syntax of a language another notation, called a meta-language, is required and in this case an extended version of Backus-Naur Form (EBNF) is used.

The syntax of the π-ADL is specified by a set of rules called productions. Each production specifies the manner in which a particular syntactic category can be formed. Syntactic categories have names which are used in productions and are distinguished from names and reserved words in the language. The syntactic categories can be mixed in productions with terminal symbols which are actual symbols of the language itself. Thus, by following the productions until terminal symbols are reached, the set of legal declarations can be derived.

The meta-symbols, that is those symbols in the meta-language used to describe the grammar of the language, include "|" which allows a choice in a production. The square brackets "[" and "]" are used in pairs to denote that a term is optional. When used with a "*", a zero or many times repetition is indicated. The reader should not confuse the meta-symbols |, *, [and] with the actual symbols and reserved words of the π-ADL. Reserved words and symbols of the π-ADL will appear in **bold.** The names of the productions will appear in normal writing. Comments are enclosed by --.

For example:

identifier ::= letter [letter | digit | _]*

indicates that an identifier can be formed as a letter, optionally followed by zero or many letters, digits or underscores.

The productions of the π-ADL are recursive, which means that there are an infinite number of legal π-ADL declarations. However, the syntax of the π-ADL can be described in a finite number of productions.

Typing Rules

The typing rules of π-ADL are used in conjunction with the context free syntax to determine the legal set of (type correct) declarations. For this a method of specifying the type rules is required.

The concept of environment needed for typing rules is introduced hereafter. Two kinds of environments are used, both of which are sets of bindings: for value identifiers and for type identifiers. δ denotes the environments where value identifiers are bound to their types in the form of <x, T>, where x is an identifier and T is a type. τ stands for the environment in which type identifiers are bound to type expressions in the form <t, T>, where t is an identifier and T is a type. A_1::b::A_2 is used to represent a list A which contains a binding b. A++B is used to denote the concatenation of two lists of bindings A and B. δ and τ are global environments and support block structure.

As introduced above, bindings are represented as pairs and the notation < x, T > is used to denote a pair value consisting of x and T. (b_1, …, b_n) is a list containing bindings b_1 to b_n. The meta function *type_declaration* takes a list of bindings between value type identifiers and value type expressions and adds them to the environment τ. Similarly, the meta function *id_declaration* takes a list of bindings between identifiers and types as its arguments and updates the environment δ with the new bindings.

The typing rules use the structure of proof rules i.e.:

$$\frac{A_1 \quad \ldots \quad A_n}{B}$$

means that if A_i is true for i = 1, . . n then B is true. Each A_i and B may be of the form X $\mapsto$ Y which, in the context of this document, is used to denote that Y is deducible from a collection of environments X. Thus, the type rule:

$$\frac{\tau, \delta \mapsto e_1 : \textbf{Integer} \quad \tau\ \delta \mapsto e_2 : \textbf{Integer}}{\tau, \delta \mapsto e_1 + e_2 : \textbf{Integer}}$$

is read as "if expression e_1 is deduced to be of type **Integer** from environments τ and δ and expression e_2 is deduced to be of type **Integer** from environments τ and δ then the type of the expression e_1 + e_2 can be deduced to be of type **Integer** from environments τ and δ.

First Class Citizenship

The application of the *Principle of Data Type Completeness* ensures that all types may be used in any combination in the language. For example, a value of any type may be a parameter of an abstraction. In addition to this, there are a number of properties possessed by all values of all types that constitute their civil rights in the language and define first class citizenship. All values of types in π-ADL have first class citizenship.

The civil rights that define first class citizenship are:

- the right to be declared,
- the right to be assigned,
- the right to have equality defined over them,
- the right to persist.

II.1 Syntax of Types

This section presents the π-ADL syntax of types. The π-ADL type system is based on the notion of types as a set structure imposed over the value space. Membership of the type sets is defined in terms of common attributes possessed by values, such as the operations defined over them. These sets or types partition the value space. The sets may be predefined, like **Integer**, or they may be formed by using one of the predefined type constructors, like **location[Integer]**.

The type constructors obey the *Principle of Data Type Completeness*. That is, where a type may be used in a constructor, any type is legal without exception. This has two benefits. Firstly, since all the rules are very general and without exceptions, a very rich type system may be described using a small number of definition rules. This reduces the complexity of the definition rules. The second benefit is that the type constructors are as powerful as possible since there are no restrictions on their domain.

Universe of Discourse

The following base types are defined in π-ADL:

The scalar data types are **Natural**, **Integer**, **Real**, and **Boolean**.

Type **String** is the type of a character string; this type embraces the empty string and single characters.

Type **Any** is an infinite union type; values of this type consist of a value of any type together with a representation of that type.

Type **Behaviour** is the type that specifies (running) behaviours. Behaviours interact with other behaviours, through connections, and pass values along these connections.

The following type constructors are defined in π-ADL:

For any type T, **connection[T]** is the type of a connection on which elements of type T can be passed.

For any type T, **location[T]** is the type of a location that contains a value of type T.

For types $T_1,...,T_n$, **tuple[T_1,...,T_n]** is the type of a tuple with elements of types T_i, for $i = 1..n$ and $n \geq 0$.

For identifiers $I_1,...,I_n$ and types $T_1,...,T_n$, **view[I_1: T_1,...,I_n: T_n]** is the type of a view with fields I_i and corresponding types T_i, for $i = 1..n$ and $n \geq 0$.

For types $T_1,...,T_n$, **union[T_1,...,T_n]** is the disjoint union of types T_i, for i = 1..n and n ≥ 0.

For identifiers $I_1,...,I_n$ and types $T_1,...,T_n$, **variant[I_1: T_1,...,I_n: T_n]** is the type of a labelled disjoint union of types: fields I_i and corresponding types T_i, for i = 1..n and n ≥ 0.

For any type T, **sequence[T]** is the type of a sequence with elements of type T.

For any type T, **set[T]** is the type of a set with elements of type T.

For any type T, **bag[T]** is the type of a bag with elements of type T.

For types $T_1,...,T_n$, **abstraction[T_1,...,T_n]** is the type of a behaviour abstraction with parameter types T_i, for i = 1..n, where n ≥ 0.

The world of data values is defined by the closure of rules **1** to **4** under the recursive application of rules **5** to **14**.

In addition to the above, declaration clauses which yield no value are of inferred type **Void**.

Each one of these types are detailed later in this document, where each particular correspondence with the abstract syntax is highlighted.

Type Algebra

π-ADL provides a simple type algebra that allows the succinct definition of types within architecture descriptions. As well as the base types and type constructors already introduced, types may be defined with the use of aliasing and recursive definitions.

Types are declared as follows.

```
type_declaration  ::=   type type_definition
          |             recursive type type_definition
                        [and type_definition]*

type_definition   ::=   identifier is type
```

Aliasing

Any legal type description may be aliased by an identifier to provide a shorthand for that type. For example:

type Int **is Integer.**
type Ints **is sequence[Integer]**

After its introduction an alias may be used in place of the full type description.

Mapping to the abstract syntax

The declaration of type identifiers is defined in the base layer π-ADL$_{BV}$ of the abstract syntax. That layer contains only base value types.

Recursive Definitions

Further expressiveness is achieved in the type algebra by the introduction of recursive types. The reserved word **recursive** introduced before a type alias allows instances of that alias to appear in the type definition. Mutually recursive types may also be defined by the grouping of aliases with **and**. In this case, binding of identifiers within the mutual recursion group takes precedence over identifiers already in scope.

```
recursive type    intList is view[head : Int, tail : realList].
and               realList is view[head : Real, tail : intList]
```

Mapping to the abstract syntax

Recursive type definitions are defined in the intermediate layer π-ADL$_{BVC}$ of the abstract syntax.

Type Equivalence

Type equivalence in π-ADL is based upon the meaning of types, and is independent of the way the type is expressed within the type algebra. Thus any aliases and recursion variables are fully factored out before equivalence is assessed. This style of type equivalence is normally referred to as structural equivalence.

The structural equivalence rules are as follows:

- Every base type is equivalent only to itself.
- For two constructed types to be equivalent, they must have the same constructor and be constructed over equivalent types.
- For view constructors the labels are a significant part of the type, but their ordering is not.
- For variant constructors the labels are a significant part of the type, but their ordering is not.
- For behaviour abstractions types, the parameter ordering is a significant part of the type, but parameter names are not.

π-ADL has no subtyping or implicit coercion rules. Values may be substituted by assignment (cf. *location*) or parameter passing only when their types are known statically to be equivalent.

The types of all expressions in π-ADL are inferred. There is no other type inference mechanism; in particular, the types of all abstraction parameters must be explicitly stated by the user.

Types

The set of legal types in π-ADL is expressed syntactically by the following production rules:

```
type      ::=     Natural
              |   Integer
              |   Real
              |   Boolean
              |   String
              |   Any
              |   connection[ [type_list] ]
              |   behaviour
              |   abstraction[ [type_list] ]
              |   tuple[ type_list ]
              |   view[ labelled_type_list ]
              |   union[ type_list ]
              |   variant[ labelled_type_list ]
              |   location[ type ]
              |   set[ type ]
              |   bag[ type ]
              |   sequence[ type ]
              |   #identifier        -- quote type
              |   identifier         -- type alias

type_list ::=     type [, type]*

labelled_type_list ::=    identifier : type [, Identifier : type]*
```

Mapping to the abstract syntax

These types correspond to the types defined in the abstract syntax. Each one of these types is detailed later on in this document.

II.2 Literals of Base Types

This section presents the syntax of π-ADL literals for base types. Literals are the basic building blocks of π-ADL declarations that allow values to be introduced.

Formally, values are behaviours accessed by names. A value *v* can be seen is a behaviour located by a name (declared in the literal and value declarations). The intuition is that a value is just a special kind of behaviour, one which can be repeatedly the subject of the same observation. Thereby, data values and behaviours are treated in a homogeneous way, even if data value abstractions and operations are provided in order to friendly deal with values.

A literal is defined by:

```
literal   ::=   natural_literal
          |     integer_literal
          |     real_literal
          |     boolean_literal
          |     string_literal
          |     connection_literal
          |     behaviour_literal
          |     abstraction_literal
          |     …
```

Mapping to the abstract syntax

The literals correspond in the abstract syntax to *values* that can be explicitly written down. The term *value* in the abstract syntax is used to make reference to values of different types.

For the base types, in the abstract syntax, the values are denoted by names, i.e. their literals. For the constructed types specific production rules are introduced.

Natural Literals

Naturals are of type **Natural** and are defined by:

natural_literal ::= **0** digit [digit]*

[natural_literal] $\dfrac{}{\tau, \delta \mapsto \mathrm{n} : \textbf{Natural}}\ \mathrm{n} \in \mathrm{Natural}$

A natural literal is one or more digits proceeded by a 0. Examples of naturals are: 03, 010, 0145.

Mapping to the abstract syntax

The typing rule for natural values is provided in the π-ADL$_{BV}$ layer of the abstract syntax.

Integer Literals

Integers are of type **Integer** and are defined by:

integer_literal	::=	[add_operator] digit [digit]*
add_operator	::=	**+** | **-**
[integer_literal]		$\frac{}{\tau, \delta \mapsto \mathrm{i} : \textbf{Integer}}\ \mathrm{i} \in \mathrm{Integer}$

An integer literal is one or more digits optionally preceded by a sign. Examples of integers are 2, -332, +67.

Mapping to the abstract syntax

The typing rule for integer values is provided in the π-ADL_{BV} layer of the abstract syntax.

Real Literals

Reals are of type **Real** and are defined by:

real_literal	::=	integer_literal.[digit]*[e integer_literal]
[real_literal]		$\frac{}{\tau, \delta \mapsto \mathrm{r} : \textbf{Real}}\ \mathrm{r} \in \mathrm{Real}$

Thus, there are a number of ways of writing a real literal. For example,

1.2	3.1e2	5.e5
1.	3.4e-2	5.4e+4

3.1e-2 means 3.1 times 10 to the power -2 (i.e. 0.031).

Mapping to the abstract syntax

The typing rule for real values is provided in the π-ADL_{BV} layer of the abstract syntax.

Boolean Literals

There are two literals of type **Boolean**: **true** and **false**. They are defined by

boolean_literal	::=	**true** | **false**

[boolean_literal] $\dfrac{}{\tau, \delta \mapsto \mathrm{b} : \textbf{Boolean}}\ \mathrm{b} \in \mathrm{Boolean}$

Mapping to the abstract syntax

The typing rule for boolean values is provided in the π-ADL_{BV} layer of the abstract syntax.

String Literals

A string literal is a sequence of characters in the character set (ASCII) enclosed by double quotes. The syntax is:

string_literal ::= "[character]*"

[string_literal] $\dfrac{}{\tau, \delta \mapsto \mathrm{s} : \textbf{String}}\ \mathrm{s} \in \mathrm{String}$

The empty string is denoted by "". Examples of other string literals are:

"This is a string literal"

To have a double quote itself inside a string literal requires using a single quote as an escape character and so if a single or double quote is required inside a string literal it must be preceded by a single quote. For example:

"a"" has the value a", and, "a'" has the value a'.

There are a number of other special characters that may be used inside string literals. They are:

'b	backspace	ASCII code 8
't	horizontal tab	ASCII code 9
'n	new line	ASCII code 10
'p	new page	ASCII code 12
'o	carriage return	ASCII code 13

Mapping to the abstract syntax

The typing rule for string values is provided in the π-ADL_{BV} layer of the abstract syntax.

Connection Literals

The connections literals are of type **connection[**T**]**, where T is a value type. They are defined by:

connection_literal ::= **connection**⟪ [type_list] ⟫

[connection_literal] $$\frac{\tau \mapsto T_1 \in \text{Type}, \ldots, T_n \in \text{Type}}{\tau, \delta \mapsto \textbf{connection(}T_1,\ldots,T_n\textbf{)} : \textbf{connection[}T_1,\ldots,T_n\textbf{]}}$$

Thus **connection(String)**, that is a connection of type **connection[String]**, has the capability of passing strings. The empty connection is noted **connection()**; it is used for pure synchronisation (formally, it passes a void value).

Mapping to the abstract syntax

Connection is one of the core concepts in π-ADL. It is the only type constructor in the base layer of the language. In the first order layer of the language, a behaviour can pass through connections any value type, including connection values. In the higher order layer, behaviours and abstractions can be passed via connections.

Behaviour Literals

There is one behaviour literal (of type **behaviour)**, used to ground the definition of behaviours.

behaviour_literal ::= **done**

[behaviour_literal] $$\frac{}{\tau, \delta \mapsto \textbf{done} : \textbf{behaviour}}$$

Behaviours will be further detailed in section II.5.

Abstraction Literals

Abstraction literals are of type ***abstraction[****Type₁, …, Type$_n$****]***, and are defined by:

abstraction_literal : := **abstraction (** [labelled_type_list] **).**clause

[abstraction_literal]

$$\frac{\tau \mapsto T_1 \in \text{Type}, \ldots \quad \tau \mapsto T_n \in \text{Type} \qquad \tau, \delta ::< x_1, T_1 >:: \ldots ::< x_n, T_n > \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{abstractio n}(x_1 : T_1, \ldots, x_n : T_n).B : \quad \textbf{abstractio n[}T_1, \ldots, T_n\textbf{]}}$$

Abstractions will be further detailed in section **Erreur ! Source du renvoi introuvable.**.

II.3 Operators of Base Types

This section presents operators of base types (**Boolean**, **Natural**, **Integer**, **Real** and **String**). It starts by addressing evaluation order. The material presented in this section is mostly a matter of concrete syntax.

Evaluation Order

The order of evaluation of a π-ADL declaration is strictly from left to right and top to bottom except where the flow of control is altered by one of the declaration clauses. Parentheses in expressions can be used to override the precedence of operators.

Parentheses

clause ::= … | expression

expression ::= **(** clause **)** | …

[parenthesis] $$\frac{\tau, \delta \mapsto \mathrm{e} : \mathrm{T}}{\tau, \delta \mapsto \mathbf{(}\mathrm{e}\mathbf{)} : \mathrm{T}}$$

These rules allow expressions in π-ADL to be written within parentheses. The effect of parentheses is to alter the order of evaluation so that the expressions in parentheses are evaluated first. For example:

```
3 * (2 - 3)
```

evaluates to -3 and not 3.

Boolean Operators

Values of type **Boolean** in π-ADL can have value **true** or **false**. There are only two boolean literals, **true** and **false**, and four operators. There is one boolean unary operator, **not**, and three boolean binary operators: **and**, **or**, **xor** and **implies**. They are defined by the truth table below:

a	*b*	***not*** *a*	*a* ***or*** *b*	*a* ***xor*** *b*	*a* ***and*** *b*	*a* ***implies*** *b*
true	**false**	**false**	**true**	**true**	**false**	**false**
false	**true**	**true**	**true**	**true**	**false**	**true**
true	**true**	**false**	**true**	**false**	**true**	**true**
false	**false**	**true**	**false**	**false**	**false**	**true**

The syntax rules for boolean expressions are:

expression ::= …
| **not** expression
| expression **or** expression
| expression **xor** expression
| expression **and** expression
| expression **implies** expression
| …

[negation] $$\frac{\tau, \delta \mapsto e : \textbf{Boolean}}{\tau, \delta \mapsto \textbf{not}\ e : \textbf{Boolean}}$$

[or] $$\frac{\tau, \delta \mapsto e_1 : \textbf{Boolean}\ \ \tau, \delta \mapsto e_2 : \textbf{Boolean}}{\tau, \delta \mapsto e_1\ \textbf{or}\ e_2 : \textbf{Boolean}}$$

[xor] $$\frac{\tau, \delta \mapsto e_1 : \textbf{Boolean}\ \ \tau, \delta \mapsto e_2 : \textbf{Boolean}}{\tau, \delta \mapsto e_1\ \textbf{xor}\ e_2 : \textbf{Boolean}}$$

[and] $$\frac{\tau, \delta \mapsto e_1 : \textbf{Boolean}\ \ \tau, \delta \mapsto e_2 : \textbf{Boolean}}{\tau, \delta \mapsto e_1\ \textbf{and}\ e_2 : \textbf{Boolean}}$$

[implies] $$\frac{\tau, \delta \mapsto e_1 : \textbf{Boolean}\ \ \tau, \delta \mapsto e_2 : \textbf{Boolean}}{\tau, \delta \mapsto e_1\ \textbf{implies}\ e_2 : \textbf{Boolean}}$$

The precedence of the operators is defined in descending order as:

not

and

or **xor**

implies

Thus,

not a **or** b **implies** c **and** d

is equivalent to

((**not** a) **or** b) **implies** (c **and** d)

The evaluation of a boolean expression in π-ADL is non-strict. That is, in the left to right evaluation of the expression, no more computation is performed on the expression than is necessary. For example,

true or *expression*

gives the value **true** without evaluating *expression* and

false and *expression*

gives the value **false** without evaluating *expression*.

Comparison Operators

Expressions of type **Boolean** can also be formed by some other binary operators. For example, a = b is either **true** or **false** and is therefore boolean. These operators are called the comparison operators and are:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
~=	not equal to

The syntactic rules for the comparison operators are:

```
expression          ::=  …
                         |  expression relational_operator expression
                         | …

relational_operator ::=  equality_operator
                         |  ordering_operator

equality_operator   ::=  =     -- equality
                         |  ~=    -- non_equality

ordering_operator   ::=  <     -- less
                         |  <=    -- less_equal
                         |  >     -- greater
                         |  =>    -- greater_equal
```

[equality]

$$\frac{\tau,\delta \mapsto e_1 : T \quad \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 = e_2 : \textbf{Boolean}}$$

[non_equality]

$$\frac{\tau,\delta \mapsto e_1 : T \quad \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 \sim= e_2 : \textbf{Boolean}}$$

In the following typing rules, T ∈ { **Natural**, **Integer**, **Real**, **String** }.

[less] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 < e_2 : \textbf{Boolean}}$$

[less_equal] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 <= e_2 : \textbf{Boolean}}$$

[greater] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 > e_2 : \textbf{Boolean}}$$

[greater_equal] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 >= e_2 : \textbf{Boolean}}$$

All the comparison operators have the same precedence.

Note that the operators <, <=, > and >= are defined on values of type **Natural**, **Integer Real** and **String**[5] whereas = and ~= are defined on all π-ADL types. The interpretation of these operations is given with each type as it is introduced.

Equality for types other than base types is defined as identity.

Arithmetic Operators

Arithmetic expressions may be evaluated on values of type **Natural**, **Integer** and **Real**. The syntax of arithmetic expressions is as follows.

```
expression          ::=   …
                    |     add_operator expression
                    |     expression  add_operator expression
                    |     expression  multiply_operator expression

multiply_operator   ::=   integer_multiply_operator
                    |     real_multiply_operator

add_operator                    ::=   +     -- plus/ add
                                |     -     -- minus/subtract

integer_multiply_operator       ::=   *     -- integer times
```

[5] The same comparison operators are used for collections (cf. section **Erreur ! Source du renvoi introuvable.**)

	| \	-- integer division
	| %	-- modulo
real_multiply_operator ::=	*	-- real times
	| /	-- real division

In the following type rules, T ∈ { **Integer**, **Real** },
T' ∈ { **Natural**, **Integer**, **Real** } and T" ∈ { **Natural**, **Integer** }.

[plus] $$\frac{\tau,\delta \mapsto e : T}{\tau,\delta \mapsto + e : T}$$

[minus] $$\frac{\tau,\delta \mapsto e : T}{\tau,\delta \mapsto - e : T}$$

[add] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 + e_2 : T'}$$

[subtract] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 - e_2 : T'}$$

[times] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 * e_2 : T'}$$

[integer division] $$\frac{\tau,\delta \mapsto e_1 : T'' \;\; \tau,\delta \mapsto e_2 : T''}{\tau,\delta \mapsto e_1 \setminus e_2 : T''}$$

[modulo] $$\frac{\tau,\delta \mapsto e_1 : T'' \;\; \tau,\delta \mapsto e_2 : T''}{\tau,\delta \mapsto e_1 \,\mathbf{\%}\, e_2 : T''}$$

[real division] $$\frac{\tau,\delta \mapsto e_1 : \mathbf{Real} \;\; \tau,\delta \mapsto e_2 : \mathbf{Real}}{\tau,\delta \mapsto e_1 / e_2 : \mathbf{Real}}$$

The operators mean:

+	addition
-	subtraction
*	multiplication
/	real division

****	integer division throwing away the remainder
%	modulo after integer division

In both **** and **%** the result is negative only if exactly one of the operands is negative.

Some examples of arithmetic expressions are:

a + b	3 + 2	1.2 + 0.5	-2.1 + a / 2.0

The language deliberately does not provide automatic coercion between naturals, integers and reals, but conversion procedures are defined in the underlying virtual machine.

Arithmetic Precedence Rules

The order of evaluation of an expression in π-ADL is from left to right and based on the precedence table:

*****	**/**	****	**%**
+	**-**		

That is, the operations *, /, \, % are always evaluated before + and -. However, if the operators are of the same precedence then the expression is evaluated left to right. For example:

6 \ 4 **%** 2 gives the value 1

Parentheses may be used to override the precedence of the operator or to clarify an expression. For example,

3 * (2 - 1) yields 3 not 5

String Operators

The string operator, ++, concatenates two operand strings to form a new string. For example:

"abc" ++ "def"

results in the string:

"abcdef"

The syntax rule is:

```
expression      ::=     …
                  |     expression ++ expression
```

[concatenation] $$\frac{\tau,\delta \mapsto e_1 : \textbf{String} \quad \tau,\delta \mapsto e_2 : \textbf{String}}{\tau,\delta \mapsto e_1 \mathbin{++} e_2 : \textbf{String}}$$

A new string may be formed by selecting a substring of an existing string. For example, if s is the string "abcdef" then s(3 | 2) is the string "cd". That is, a new string is formed by selecting 2 characters from s starting at character 3. The syntax rule is:

expression ::= …
| expression **(** clause **|** clause **)**

[substring] $$\frac{\tau,\delta \mapsto e : \textbf{String} \quad \tau,\delta \mapsto e_1 : \textbf{Natural} \quad \tau,\delta \mapsto e_2 : \textbf{Natural}}{\tau,\delta \mapsto e\,(e_1 \mid e_2) : \textbf{String}}$$

For the purposes of substring selection the first character in a string is numbered 1. The selection values are the start position and the length respectively.

To compare two strings, the characters are compared in pairs, one from each string, from left to right. Two strings are considered equal only if they have the same characters in the same order and are of the same length, otherwise they are not equal.

The characters in a string are ordered according to the ASCII character code. Thus:

"a" < "z"

is true.

The empty string "" is less than any other string. Thus the less-than relation can be resolved by taking the characters pair by pair in the two strings until one is found to be less than the other. When the strings are not of equal length then they are compared as above and then the shorter one is considered to be less that the longer. Thus:

"abc" < "abcd"

The other relations can be defined by using = , < and ~.

Precedence Table

The full precedence table for operators in π-ADL is:

/ * **** **%**

\+ - ++

~

= ~= < <= > >=

and

or **xor**

implies

II.4 Value and Type Declarations

All π-ADL declarations (of values and types) as well as declarations that composes behaviours[6] are of inferred type **void**. Grouped together through behaviour constructs, they can compose behaviours (of type **behaviour**). This is defined using typing rules. This section presents value and type declarations.

Identifiers

In π-ADL, an identifier may be bound to a value, an abstraction parameter, a view or variant field, or a type. An identifier may be formed according to the following syntactic rule:

identifier ::= letter [letter | digit | _]*

That is, an identifier consists of a letter followed by any number of letters, digits or underscores. The following are legal π-ADL identifiers:

x1 oneValue Look_for_record aComponent

Note that case is significant in identifiers.

The use of an identifier is governed by the syntactic rule:

expression ::= … | identifier

The type rule states that the type of an identifier can be deduced from the value environment δ.

[identifier] $\overline{\tau, \delta_1 ::< x, T >:: \delta_2 \mapsto x : T}$

[6] These are clauses in the concrete syntax. They are presented later on in this document (cf. section II.5)

Declaration of Value Identifiers

Before an identifier can be used in π-ADL, it must be declared. The action of declaring a value associates an identifier with a typed value.

When introducing a value identifier, the identifier and its value must be declared. Identifiers are declared using the following syntax.

value_declaration ::= **value** identifier_clause_list
| …

identifier_clause_list ::= identifier **is** clause [**.** identifier **is** clause]

[value declaration]

$$\frac{\tau \mapsto T \in \text{Type} \quad \tau, \delta \mapsto e : T}{\tau, \delta \mapsto \textbf{value}\ \text{name}\ \textbf{is}\ e\ :\ \textbf{void}\quad \text{id\_declaration}(<\text{name}, T>)}$$

For example:

value a **is** 1

introduces an integer identifier *a* with value 1. The type Integer is inferred.

Note that the declaration of a value identifier induces the introduction in the environment of a binding between the identifier and its type (using the meta function *id_declaration*).

Mapping to the abstract syntax

Declaration of value identifiers is defined in the first order layer of the abstract syntax.

Declaration of Types

Type names may be explicitly declared in π-ADL. The name is used to represent a set of values drawn from the value space and may be used wherever a type identifier is legal. The syntax of type declarations, i.e. aliasing, is:

type_declaration ::= **type** type_definition
| …

type_definition ::= identifier **is** type

[type declaration]

$$\frac{\tau \mapsto T \in \text{Type}}{\tau, \delta \mapsto \textbf{type}\ \text{name}\ \ \textbf{is}\ T\ :\ \textbf{void}\quad \text{type\_declaration}(<\text{name}, T>)}$$

Thus:

type B **is Boolean**

is a type declaration aliasing the identifier *B* with the type **Boolean**. They are the same type and may be used interchangeably.

Mapping to the abstract syntax

Declaration of type identifiers is defined in the first order layer of the abstract syntax. As in the abstract syntax, in the concrete syntax declarations of types and values are expressed in the context of behaviours or behaviour abstractions.

Descriptions

A description in π-ADL is composed of any combination, in any order, of value and type declarations and declaration clauses. The type of the description is the type of the last clause in the description. If there is more than one clause in a description then all but the last must be of type **void**. Where the description ends with a declaration, which by definition is of type **void**, the description is of type **void**. If the description ends with a clause corresponding to the behaviour literal **done**, the declaration has the type **behaviour**. This is how a combination of clauses constructs a behaviour.

description ::= declaration [**.** description]
| clause [**.** description]

[declarations]

$$\frac{\tau,\delta \mapsto D_1 : \textbf{void} \quad \text{type\_declaration}(\Omega_1) \quad \text{id\_declaration}(\Psi_1) \quad \tau{+}{+}\Omega_1, \delta{+}{+}\Psi_1 \mapsto D_2 : T}{\tau,\delta \mapsto D_1.D_2 : T}$$

Brackets

Brackets are used to make a description of clauses and declarations into a single clause.

expression ::= **{** description **}**
| …

[{}]
$$\frac{\tau,\delta \mapsto e : T}{\tau,\delta \mapsto \{e\} : T}$$

This is mainly used to delimit behaviours.

Scope Rules

The scope of an identifier is limited to the rest of the description following its declaration (i.e. it follows the brackets structure). If the same identifier is declared in an inner description, then while the inner name is in scope the outer one is not. Connections, however, may having changing scopes. That is, scope extrusion may dynamically extrude scope of restricted connections by sending them via connections to other behaviours out of their actual scope (restriction is a static binder).

Typed identifiers can be explicitly introduced in descriptions (it mainly used for declaring connection names):

clause ::= **name** identifier_type_list
| …

Recursive Value Declarations

It is sometimes necessary to define values recursively. The initialising clauses for recursive declarations are restricted to literal values.

The full syntax of value declarations is:

value_declaration ::= **value** identifier_clause_list
| **recursive value** identifier_literal_list

identifier_clause_list ::= identifier **is** clause [**.** identifier **is** clause]

identifier_literal_list ::= identifier **is** literal [**and** identifier **is** literal]*

[value declaration]

$$\frac{\tau \mapsto T \in \text{Type} \quad \tau, \delta \mapsto e : T}{\tau, \delta \mapsto \textbf{value}\ \text{name}\ \textbf{is}\ e\ :\ \textbf{void}\quad \text{id\_declaration}(<\text{name}, T>)}$$

[recursive value declaration]

$$\frac{\tau, \delta \mapsto e_1 : T_1 \quad \tau, \delta \mapsto e_2 : T_2}{\tau, \delta' \mapsto \textbf{recursive value}\ i_1 = e_1\ \textbf{and}\ i_2 = e_2 : \textbf{void}\ \text{id\_declaration}(< i_1, T_1 >, < i_2, T_2 >)}$$

where δ stands for δ1::δ2::δ3 and
δ' stands for δ1::< i1, T1 >::δ2::< i2, T2 >::δ3

Recursive Type Declarations

The full syntax of type declarations is:

```
type_declaration  ::=   type definition
          |             recursive type type_definition
                        [and type_definition]*

type_definition   ::=   identifier is type
```

[type declaration]

$$\frac{\tau \mapsto T \in \text{Type}}{\tau, \delta \mapsto \textbf{type}\ \text{name}\ \textbf{is}\ T : \textbf{void}\quad \text{type\_declaration}(<\text{name}, T>)}$$

[recursive type declaration]

$$\frac{\tau \mapsto T_1 \in \text{Type} \quad \tau \mapsto T_2 \in \text{Type}}{\tau' \mapsto \textbf{recursive type}\ t_1\ \textbf{is}\ T_1\ \textbf{and}\ t_2\ \textbf{is}\ T_2 : \textbf{void}\ \text{type\_declaration}(<t_1, T_1>, <t_2, T_2>)}$$

where τ stands for $\tau_1::\tau_2::\tau_3$ and τ' stands for $\tau_1::<t_1, T_1>::\tau_2::<t_2, T_2>::\tau_3$

Mapping to the abstract syntax

Recursive type definitions are introduced in the first order layer π-ADL.

II.5 Clauses for Declaring Behaviours

Expressions are clauses which allow operators in the language to be used to produce values. There are other kinds of clauses in π-ADL which allow the values to be manipulated (like changing the value of a location, projecting values that are composites). But what most characterises clauses is the fact that they supports the declaration of behaviours. This section presents declaration clauses for behaviours.

Prefix

Prefix expresses the capability of sending and receiving values via connections, enacting unobservable actions, enacting conditional actions, and restring connections. Prefixes are defined as follows.

```
clause  ::=   …
          |   prefix . clause  …                                  -- prefix

prefix  ::=   via expression send [clause}                       -- output prefix
          |   via expression receive [identifier_type_list]      -- input prefix
          |   unobservable                                       -- silent prefix
          |   restrict name identifier_type_list

identifier_type_list      ::=   identifier [: type] [, identifier [: type]]*
```

[unobservable] $$\frac{}{\tau,\delta \mapsto \textbf{unobservable} : \textbf{void}}$$

[receive 1] $$\frac{\tau \mapsto \mathrm{T} \in \mathrm{Type}\ \tau,\delta \mapsto \mathrm{name} : \textbf{connection}[\mathrm{T}]\ \ \tau,\delta \mapsto \mathrm{x} : \mathrm{T}}{\tau,\delta \mapsto \textbf{via}\ \mathrm{name}\ \textbf{receive}\ \mathrm{x} : \textbf{void}}$$

[receive 2] $$\frac{\tau \mapsto \mathrm{T} \in \mathrm{Type}\ \tau,\delta \mapsto \mathrm{name} : \textbf{connection}[\mathrm{T}]}{\tau,\delta \mapsto \textbf{via}\ \mathrm{name}\ \textbf{receive}\ \mathrm{x} : \mathrm{T} : \textbf{void}\ \ \mathrm{id\_declaration}(<\mathrm{x},\mathrm{T}>)}$$

[send] $$\frac{\tau \mapsto \mathrm{T} \in \mathrm{Type}\quad \tau,\delta \mapsto \mathrm{name} : \textbf{connection}[\mathrm{T}]\ \ \tau,\delta \mapsto \mathrm{x} : \mathrm{T}}{\tau,\delta \mapsto \textbf{via}\ \mathrm{name}\ \textbf{send}\ \mathrm{x} : \textbf{void}}$$

[restriction] $$\frac{\begin{array}{c}\tau \mapsto \mathrm{T}_1 \in \mathrm{Connection\ Type}\ \ldots\ \tau \mapsto \mathrm{T}_n \in \mathrm{Connection\ Type} \\ \tau,\delta :: <\mathrm{i}_1,\mathrm{T}_1> :: \ldots :: <\mathrm{i}_n,\mathrm{T}_n> \mapsto \mathrm{B} : \textbf{behaviour}\end{array}}{\tau,\delta \mapsto \textbf{restrict}\ \ \mathrm{i}_1 : \mathrm{T}_1\textbf{,...,}\ \mathrm{i}_n : \mathrm{T}_n\ \textbf{.}\ \mathrm{B} : \textbf{behaviour}\ \ \mathrm{id\_declaration}(<\mathrm{i}_1,\mathrm{T}_1>,\ldots,<\mathrm{i}_n,\mathrm{T}_n>)}$$

A value that is sent has to be previously typed (cf. type rule [*send*]). For the reception of values, the type may be inferred (cf. type rule [*receive 1*]), or explicitly declared (cf. type rule [*receive 2*]). Restriction expresses the capability to restrict the scope of a name to the scope of a behaviour. It mainly used for restricting connection names.

Mapping to the abstract syntax

Prefix is part of the base layer of π-ADL. In the abstract syntax, the prefix production rule indicates that a prefix always precedes a behaviour. The type rule is considered accordingly, thus the declaration of a prefix is part of an expression of type behaviour. In the concrete syntax, a prefix itself is of type **void**. The prefix is then integrated in expression of behaviour through the clause production rule.

The match prefix in the abstract syntax is defined hereafter in the concrete syntax.

Matching Prefix

The matching prefix ***if*** *condition* ***do*** *behaviour* indicates a behaviour that will enact only if the condition is verified. A variant, using the choice operator is defined with the condition and its negation: ***if*** *condition* ***then*** *behaviour1* ***else*** *Behaviour2*.

The concrete syntax is given as follows.

```
clause    ::=    …
```

	if expression **do** clause	-- matching prefix
	if expression **then** clause **else** clause	

[matching prefix] $$\frac{\tau, \delta \mapsto \text{expr} : \textbf{Boolean} \quad \tau, \delta \mapsto \text{B} : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{if}\ \text{expr}\ \textbf{do}\ \text{B} : \textbf{behaviour}}$$

Mapping to the abstract syntax

The matching prefix is part of the base layer of π-ADL.

Choice

Choice **choose {***Behaviour*$_1$ **or** *Behaviour*$_2$ **or** … **or** *Behaviour*$_n$**}** expresses the capability of a behaviour to choose the capability of *Behaviour*$_1$ or the capability of *Behaviour*$_2$*, etc.*. When one of the capabilities is exercised, the others are no longer available. Thereby, the choice will proceed as a behaviour *Behaviour*$_i$*'* after exercising the capability of *Behaviour*$_i$.

The concrete syntax is as follows.

clause	::=	…
	|	**choose {** [choice_list] **}** -- choice
choice_list	::=	clause [**or** clause]*

[choice] $$\frac{\tau, \delta \mapsto \text{B}_1 : \textbf{behaviour} \quad \tau, \delta \mapsto \text{B}_2 : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{choose}\{\text{B}_1\ \textbf{or}\ \text{B}_2\} : \textbf{behaviour}}$$

Mapping to the abstract syntax

The choice construct is part of the base layer of π-ADL.

Composition

Composition **compose {***Behaviour*$_1$ **and** *Behaviour*$_2$ **and**… **and** *Behaviour*$_n$**]** expresses the capability of a behaviour to parallely compose the capabilities of *n* given behaviours *Behaviour*$_1$ … *Behaviour*$_{n,}$. *Behaviour*$_i$ can proceed independently and can interact via shared connections. Thereby, the composition will proceed by exercising the capabilities of independent actions in *Behaviour*$_1$*'* and *Behaviour*$_2$*'* and so on, or jointly exercising a capability of interaction, i.e. when via a shared connection one behaviour sends a value and another one receives the value yielding an unobservable communication action.

The concrete definition is given by :

clause	::=	…

	|	**compose {** [parallel_list] **}** -- composition
parallel_list	::=	clause [**and** clause]*
[composition]		$\dfrac{\tau,\delta \mapsto B_1 : \textbf{behaviour} \quad \tau,\delta \mapsto B_2 : \textbf{behaviour}}{\tau,\delta \mapsto \textbf{compose}\{B_1 \textbf{ and } B_2\} : \textbf{behaviour}}$

Mapping to the abstract syntax

The compose construct is part of the base layer of π-ADL.

Replication

Replication ***replicate*** *Behaviour* expresses the capability of a behaviour to replicate itself infinitely. It can be thought of as an infinite composition **compose {***Behaviour* ***and*** *Behaviour* ***and*** *…***}** or **compose {***Behaviour* ***and replicate*** *Behaviour***}**.

The concrete syntax is as follows.

clause	::=	…
	|	**replicate** clause -- replication
[replication]		$\dfrac{\tau,\delta \mapsto B : \textbf{behaviour}}{\tau,\delta \mapsto \textbf{replicate } B : \textbf{behaviour}}$

Mapping to the abstract syntax

The replication construct is part of the base layer of π-ADL.

Renaming

Renaming means applying a map that substitutes names in a behaviour by new names.

The concrete syntax is as follows.

expression	::=	…
	|	clause **where {** renaming_list **}**
renaming_list	::=	name **renames** name [**.** name **renames** name]*

[renaming]

$$\frac{\tau \mapsto \text{T1},\ldots,\text{Tn} \in \text{ConnectionType} \quad \tau,\delta \mapsto \text{e} : \textbf{behaviour}}{\tau,\delta \mapsto \text{e } \textbf{where } \{\text{x } \textbf{renames } \text{y}\} : \textbf{void } \text{id\_substitution}(< x, \text{y} >)}$$

Given the renaming construct, α-convertibility of behaviours can be defined. If a name *w* does not occur in a behaviour *B*, then *B* ***where {****w* ***renames*** *z****}*** is the behaviour obtained by replacing each free occurrence of *z* in *B* by *w*.

A name itself can be composed of several constituents, syntactically separated by **::**, yielding prefixed names. Prefixed names can be manipulated using pattern matching.

Mapping to the abstract syntax

The renaming is part of the base layer of π-ADL.

Running Behaviours

In order to express running behaviours, the keyword **behaviour** is used. It launches the evaluation of the behaviour expression by the virtual machine.

running_behaviour	::=	**behaviour** clause

II.6 Declaring Abstractions

This section presents declaration clauses for abstractions.

Abstractions

An abstraction is a, possibly structured, parameterised behaviour. It is useful to define parameterised behaviours and recursive behaviours[7]. In π-ADL, this facility is provided through the abstraction construct: ***abstraction*** *BehaviourAbstraction* where a behaviour abstraction is expressed by **(** $x_1,\ldots,x_n$ **).***Behaviour* where *free names*(*Behaviour*) $\subseteq \{x_1,\ldots,x_n\}$ and *bound names*(*Behaviour*) $\cap \{x_1,\ldots,x_n\} = \varnothing$.

Formally, in the first order layer, abstractions are defined as replicated input behaviours. In order to apply abstractions, the send construct is used. Thereby:

value name **is abstraction (**$x_1,\ldots,x_n$**).**Behaviour

is formally defined as:

replicate via name **receive (**$x_1,\ldots,x_n$**).**Behaviour

[7] Abstractions (including ecursive definition of behaviours) can be obtained using replication.

where name is a fresh name.

To apply an abstraction, the send construct is used:

via name **send (**v_1,…,v_n**).**Behaviour

In a recursive definition, the abstraction is defined in terms of itself, i.e. using applications of itself.

recursive value name **is abstraction (**x_1,…,x_n**).**Behaviour

where applications of itself is used in Behaviour.

The concrete syntax is as follows.

abstraction_literal ::= **abstraction (** [labelled_type_list] **).** clause

[abstraction literal]

$$\frac{\tau \mapsto T_1 \in \text{Type}, \ldots, \tau \mapsto T_n \in \text{Type} \quad \tau, \delta :: < x_1, T_1 > :: \ldots :: < x_n, T_n > \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{abstraction}(x_1 : T_1, \ldots, x_n : T_n).B : \textbf{abstraction}[T_1, \ldots, T_n]}$$

Syntactically[8], an abstraction declared using a single name has automatically a prefix form, in which it can be written before its parameters enclosed in parenthesis. If such a declaration contains names beginning with an underscore character '_' then it must contain exactly as many underscored names as the number of its parameters, and in that case it has also a mixfix form, with a composite name. An abstraction declaration can also be named with several non-underscored names, but semantically it corresponds to a single name. In order to apply abstractions, applications may be applied in a mixfix form.

The concrete syntax for abstractions is extended with composite names as follows.

abstraction_mixfix ::= **definition value** [identifier* [identifier : type]]* **.** clause

Mapping to the abstract syntax

Abstractions are part of the base layer of π-ADL.

8 This is not part of the abstract syntax, but introduced for readability purposes.

Partial application of abstractions

In a partial application of abstractions not all parameters are given a value, i.e. a v_i is not provided for every x_i. Such application yields another abstraction. The parameters of the new abstraction are the ones for which a value was not provided.

[abstraction derivation]

$$\frac{\text{Decl1}\quad \tau,\delta \mapsto A : \textbf{behaviour[}T_1,\ldots,T_n\textbf{]}\ \tau,\delta \mapsto x_{pi1} : T_{i1},\ldots,\tau,\delta \mapsto x_{pik} : T_{ik}}{\tau,\delta \mapsto A(x_{pi1}\textbf{ renames } x_{i1},\ldots,x_{pik}\ \textbf{...}\ x_{ik}) : \textbf{behaviour[}T_{j1},\ldots,T_{jl}\textbf{]}\quad \text{Decl2}}$$

where:

Decl1 stands for:

$\tau \mapsto T_1 \in \text{Type},\ldots,\tau \mapsto T_n \in \text{Type}$

Decl2 stands for:

$\text{id\_declaration}(< x_{pi1} : T_{i1} >,\ldots,< x_{pik} : T_{ik} >)\ \text{and}\ \{T_{i1},\ldots,T_{ik}, T_{j1},\ldots,T_{jl}\} = \{T_1,\ldots,T_n\}, k < n$

II.7 Declaring Locations

Values may be stored in locations and subsequently retrieved.

Construction and dereference

The constructor **locate** creates a location and initialises the value in it. The operator **'** (dereference) retrieves the value from the location. Since locations are values in π-ADL they may also be stored in locations. The syntactic rules are:

```
expression      ::=   …
                  |   via expression locate expression -- make location
                  |   'expression                      -- dereference location
```

[location_value]

$$\frac{\tau \mapsto T \in \text{ValueType}\quad \tau,\delta \mapsto v : T}{\tau,\delta \mapsto \textbf{location}(v) : \textbf{location[}T\textbf{]}}$$

[location_dereference]

$$\frac{\tau \mapsto T \in \text{ValueType}\quad \tau,\delta \mapsto l : \textbf{location[}T\textbf{]}}{\tau,\delta \mapsto\ 'l : T}$$

For example, if *a* is of type **location[Integer]** with the value **location(**3**)** then *'a* has the value 3.

Locations are manipulated through connections (formally, it corresponds to a send action):

```
via gamma locate ( b - 4.0 * a)
```

stores the calculated real value in the location. The value in the location may be updated by re-locating a new value:

via gamma **locate** ('gamma * 2.5)

Another example:

via discriminant **locate** (0.0)**.**

via discriminant **locate** (b * b - 4.0 * a * c)

gives *discriminant* the value of the expression. The clause alters the value in the location denoted by the identifier.

The semantics of location is defined in terms of equality. The clause:

via a **locate** b

where *a* and *b* are both identifiers, implies that after execution *'a = b* will be true. For scalar types, it means value equality and for constructed types it means identity.

Equality and Equivalence

Two locations are equal if they have the same identity, that is, the same location. Two locations are type equivalent if they have equivalent content types.

Mapping to the abstract syntax

Location is provided in the π-ADL_{BV} layer of the abstract syntax.

II.8 Values of Composite Types

π-ADL allows to group together values into larger aggregate values which may then be treated as single values. There are several such value types in π-ADL:

- tuple,
- view,
- union,
- any,
- quote,
- variant,
- set,
- bag,
- sequence.

Each type construct introduces a new set of composite values[9], which can be decomposed into their elements by means of pattern matching. Each form of pattern matching is a new behaviour construct, with associated operational rules in which the decomposition of a value generates a τ action.

Composite values have the same civil rights as any other data value in π-ADL.

Mapping to the abstract syntax

In the abstract syntax the types tuple, view, union, any, quote and variant are introduced in the layer π-ADL$_{BVC}$, while the collection types (i.e. sequence, set and bag) are introduced in the layer π-ADL$_{BVCC}$. The rest of the section presents each of these types individually.

Literals of Composite Types

A composite literal is defined by:

```
literal   ::=   …
          |     tuple_literal
          |     view_literal
          |     union_literal
          |     variant_literal
          |     set_literal
          |     bag_literal
          |     sequence_literal
```

For each composite type, its empty value and other literals are defined hereafter.

Tuples

The values of a tuple type **tuple[**T_1, …, T_n**]** are n-tuples **tuple(**v_1, …, v_n**)** where each v_i is of type T_i.

Tuple values are broken into their elements by the *project* construct: **project** v **as** x_1, …, x_n.

```
type          ::=   …
                    | tuple[ [type_list] ]

type_list     ::=   type [, type]*

expression    ::=   ….
```

[9] All aggregate values in π-ADL have "pointer" semantics. That is, when an aggregate value is created, a pointer to the aggregate that makes up the value is also created. The value is always referred to by the pointer, which may be passed around by location and tested for equality.

	|	**project** identifier **as** identifier_list
	|	**tuple(** [clause_list] **)** -- make tuple

[tuple value]
$$\frac{\tau \mapsto T1,\ldots,Tn \in \text{Type} \quad \tau,\delta \mapsto e_1 : T_1 \ldots \tau,\delta \mapsto e_n : T_n}{\tau,\delta \mapsto \textbf{tuple}(e_1,\ldots,e_n) : \textbf{tuple}[T_1,\ldots,T_n]}$$

[tuple dereference]
$$\frac{\tau \mapsto T1,\ldots,Tn \in \text{ValueType} \quad \tau,\delta \mapsto e : \textbf{tuple}[T_1,\ldots,T_n]}{\tau,\delta \mapsto \textbf{project}\ v\ \textbf{as}\ x_1,\ldots,x_n \quad e : \textbf{void} \quad \text{id\_declaration}(< x_1, T_1 >,\ldots,< x_n, T_n >)}$$

Mapping to the abstract syntax

Tuples are part of the first order layer of π-ADL. The mapping is straightforward.

Views

Views are labelled forms of tuples. The values of a view type **view[**$label_1 : T_1$, …, $label_n : T_n$**]** are views **views(**$label_1 = v_1$, …, $label_n = v_n$**)** where each v_i is of type T_i.

To obtain a field of a view, the field label is used as an index. The syntax is *v::label*$_i$.

type	::=	…
	|	**view[** [labelled_type_list] **]**
labelled_type_list	::=	identifier **:** type [**,** identifier **:** type]*
expression	::=	…
	|	clause**::**identifier -- dereference view
	|	**view(** [identifier_clause_list] **)** -- make view

[view value]
$$\frac{\tau \mapsto T1,\ldots,Tn \in \text{Type} \quad \tau,\delta \mapsto v_1 : T_1 \ \ldots \ \tau,\delta \mapsto v_n : T_n}{\tau,\delta \mapsto \textbf{view}\,(l_1 : v_1,\ldots,l_n : v_n) : \textbf{view[}l_1 : T_1,\ldots,l_n : T_n\textbf{]}}$$

[view dereference]
$$\frac{\tau \mapsto T1,\ldots,Tn \in \text{Type} \quad \tau,\delta \mapsto v : \textbf{view[}l_1 : T_1,\ldots,l_n : T_n\textbf{]}}{\tau,\delta \mapsto v :: l_i \textbf{ :} T_i}$$

Mapping to the abstract syntax

Views are part of the first order layer of π-ADL. The mapping is straightforward. In the concrete syntax, each of the constituent elements of the view can be accessed independently by dereference (a shorthand for the project construct).

Unions

A union type **union[**$T_1,\ldots,T_n$**]** is the disjoint union of the types $T_1, \ldots, T_n$. The values of a union type **union[**$T_1, \ldots, T_n$**]** are unions **union(**T_i**::**v_i**)**.

Union values are broken into their elements by the *project* construct:

project e **as** x **. case {**T_1 **do** e_1 **or** … **or** T_n **do** e_n **or default do** e_{n+1}**}**

where *x* takes the value *e*. If the inferred type of *x* is one of types T_i expression e_i is executed (this expression may use *x*). The ***default*** is chosen if the inferred type is not one of types T_i, $1 \le i \le n$.

type ::= …
| **union[** [type_list] **]**

type_list ::= type [**,** type_list]

expression ::= …
| **union(**[type**::**]clause**) :** type

clause ::= …
| **project** clause **as** identifier **. case {** project_list **}**

[union-i value]

$$\frac{\tau, \delta \mapsto v : T_i}{\tau_1 ::< T, \textbf{union[}T_1 \ldots T_n\textbf{]} >:: \tau_2, \delta \mapsto \textbf{union}(T_i : v) : T : \textbf{union[}T_1 \ldots T_n\textbf{]}}$$

[union projection]

$$\frac{\tau, \delta \mapsto e : \textbf{union[}T_1, \ldots, T_n\textbf{]} \quad \forall i \in \{1,\ldots,n+1\} (\tau, \delta_1 ::< x, T_i >:: \delta_2 \mapsto e_i : \textbf{T})}{\tau, \delta \mapsto \textbf{project } e \textbf{ as } x \textbf{ case } \{T_1 \textbf{ do } e_1; \textbf{ or} \ldots \textbf{or } T_n \textbf{ do } e_n \textbf{ or default } e_{n+1}\} : T \quad \text{id\_declaration}(< x, T_i >)}$$

Mapping to the abstract syntax

Unions are part of the first order layer of π-ADL. The mapping is straightforward. The values of a union type **union[**$T_1,\ldots,T_n$**]** are of the form $tag_1::v, \ldots, tag_n::v$ where $tag_1, \ldots, tag_n$ are tags that tell which of the types $T_1, \ldots, T_n$ the value *v* comes from. Syntactically, a value being typed, the type can implicitly tag a union value: **union(**v**)** where *v* is of type T_i, its tag in the union value.

Any

The type ***Any*** is an infinite union type. Any values are broken into their elements by the *project* construct:

project e **as** x **. case {**T_1 **do** e_1; …, T_n **do** e_n **or default do** e_{n+1}**}**

where *x* takes the value *e*. If the inferred type of *x* is one of types T_i expression e_i is executed (this expression may use *x*). The ***default*** is chosen if the inferred type is not one of types T_i, $1 \leq i \leq n$.

type ::= …
| **Any** -- Any is a base type

type_list ::= type [**,** type_list]

expression ::= …
| **any(**[type**::**]clause**)**

clause ::= …
| **project** clause **as** identifier **. case {** project_list **}**

[any-i value]

$$\frac{\tau, \delta \mapsto v : T_i}{\tau_1 ::< T, \textbf{union[}T_1 \ldots T_n\textbf{]} >:: \tau_2, \delta \mapsto \textbf{union}(T_i : v) : T : \textbf{union[}T_1 \ldots T_n\textbf{]}}$$

[any-injection] $$\frac{\tau \mapsto T \in Type \qquad \tau, \delta \mapsto v : T}{\tau, \delta \mapsto \textbf{any}(v) : \textbf{Any}}$$

[any projection]

$$\frac{\tau \mapsto T1, \ldots, Tn \in ValueType \qquad \tau, \delta \mapsto v : \textbf{Any} \qquad \forall i \in \{1, \ldots, n+1\} (\tau, \delta_1 ::< x, T_i >:: \delta_2 \mapsto e_i : \textbf{T)}}{\tau, \delta \mapsto \textbf{project } e \textbf{ as } x . \textbf{ case } \{T_1 \textbf{ do } e_1 \textbf{ or } \ldots \textbf{or } Tn \textbf{ do } en \textbf{ or default do } e_{n+1}\} : T \quad \text{id\_declaration}(< x, T_i}$$

Mapping to the abstract syntax

Any is part of the first order layer of π-ADL. The mapping is straightforward.

Quotes

A quote type has only one element, and this has the same representation as the type itself. The combination of union type and quote types is similar to what is known as an enumerated type.

The rules defining the quote type and values are the following:

type ::= …
| #identifier -- quote literal

[quote literal] $$\frac{i \in Identifier}{\tau \mapsto i : \#i}$$

Mapping to the abstract syntax

Quotes are part of the first order layer of π-ADL. The mapping is straightforward.

Variants

Variants are to union types what views are to tuples: labelled disjoint unions of types.

A variant type **variant[**label$_1$: T$_1$,…, label$_n$: T$_n$**]** is the labelled disjoint union of the types *T$_1$, …,T$_n$*, labelled by *label$_1$,…, label$_n$*. The values of a variant type **variant[**label$_1$: T$_1$,…, label$_n$: T$_n$**]** are labelled unions **variant (**label$_i$**:** v$_i$**)** : T where *T* is an alias of the variant type (i.e. of **variant[**label$_1$: T$_1$,…, label$_n$: T$_n$**]**) and v_i is of one of the types T_i according to *label$_i$*.

Variant values are broken into their elements by the *project* construct:

project e **as** x **. case {**label$_1$ **do** e$_1$ **or** … **or** label$_n$ **do** e$_n$ **or default do** e$_{n+1}$**}**

where if *l* is one of the labels *label$_i$*, the expression e_i is executed (this expression may use *v*). The ***default*** is chosen when *l* is not one of the labels types *label$_i$* ,*1≤i≤n.*

The rules defining the variant type and values are the following:

type	::=	…
	|	**variant[** [labelled_type_list] **]**
labelled_type_list	::=	identifier **:** type [**,** identifier **:** type]*
expression	::=	…
	|	**variant(** identifier **::** clause **)** : type

[variant value]

$$\frac{\tau \mapsto T1,\ldots,Tn \in \text{Type} \qquad \tau,\delta \mapsto v : T_i}{\tau_1 ::< T, \mathbf{variant[}l_1 : T_1,\ldots,l_i : T_i,\ldots,l_n : T_n\mathbf{]} >:: \tau_2,\delta \mapsto \mathbf{variant}(l_i :: v) : T : \mathbf{variant[}l_1 : T_1,\ldots,l_i : T_i,\ldots,l_n : T_n\mathbf{]}}$$

[variant projection]

$$\frac{\tau,\delta \mapsto e : \mathbf{variant[}l_1 : T_1,\ldots,l_i : T_i,\ldots,l_n : T_n\mathbf{]} \quad \forall i \in \{1,\ldots,n+1\}(\tau,\delta_1 ::< x,T_i >:: \delta_2 \mapsto e_i : \mathbf{T})}{\tau,\delta \mapsto \mathbf{project}\ e\ \mathbf{as}\ x\ \mathbf{.\ case}\ \{l_1\ \mathbf{do}\ e_1\ \mathbf{or}\ \ldots\mathbf{or}\ l_n\ \mathbf{do}\ e_n\ \mathbf{or\ default\ do}\ e_{n+1}\} : T \quad \text{id\_declaration}(< x,T_i >)}$$

Mapping to the abstract syntax

Variants are part of the first order layer of π-ADL. The mapping is straightforward.

Sets

A set is a collection, which contains elements without, duplicates and that are not ordered. The values of a set type **set[**T**]** are sets **set(**v_1, …, v_n**)** where each v_i is of type *T*. The empty set is written: **set()**.

The rules defining the set type and values are the following:

```
type            ::=   …
                  |   set[ type ]

expression      ::=   …
                  |   set( [clause_list] )          -- make set
                  |   iterate expression [by identifier : type]
                      accumulate clause
                      [as identifier]
                  |   iterate expression [by identifier : type]
                      do clause
                  |   expression includes expression
                  |   expression excludes expression

set_literal ::=   set()
                  τ,δ ↦ e1 : T ... τ,δ ↦ en : T
[set value]       ---------------------------------
                  τ,δ ↦ set( e1,...,en ) : set[ T ]
```

Mapping to the abstract syntax

The mapping between concrete syntax and abstract syntax is straightforward.

Bags

A bag is a collection, which contains elements with duplicates allowed and that are not ordered. An element may be part of a bag more than once. The values of a bag type **bag[**T**]** are bags **bag(**v_1, …, v_n**)** where each v_i is of type *T*. The empty bag is written: **bag()**.

The rules defining the bag type and values are the following:

```
type            ::=   …
                  |   bag[ type ]

expression      ::=   …
                  |   bag( [clause_list] )          -- make set
                  |   iterate expression [by identifier : type]
                      accumulate clause
```

		[**as** identifier]
	|	**iterate** expression [**by** identifier **:** type]
		do clause
	|	expression **includes** expression
	|	expression **excludes** expression
	|	expression **excludes all** expression
bag_literal	::=	**bag()**
[bag value]		$\frac{\tau,\delta \mapsto e_1 : T \ldots \tau,\delta \mapsto e_n : T}{\tau,\delta \mapsto \textbf{bag}(e_1,\ldots,e_n) : \textbf{bag}[T]}$

Mapping to the abstract syntax

The mapping between concrete syntax and abstract syntax is straightforward.

Sequences

A sequence is a collection in which the elements are ordered. An element may be part of a sequence more than once. The values of a sequence type **sequence[**T**]** are sequences **sequence(**v_1, …, v_n**)** where each v_i is of type *T*. The empty sequence is written: **sequence()**.

The rules defining the set type and values are the following:

type	::=	…
	|	**sequence[** type **]**
expression	::=	…
	|	**sequence(** [sequence_clause_list] **)** -- make sequence
	|	**iterate** expression [**by** identifier **:** type]
		accumulate clause
		[**as** identifier]
	|	**iterate** expression [**by** identifier **:** type]
		do clause
	|	expression **includes** expression
	|	expression **excludes** expression
sequence_clause_list	::=	clause_list
	|	expression**..**expression -- range
[sequence value]		$\frac{\tau,\delta \mapsto e_1 : T \ldots \tau,\delta \mapsto e_n : T}{\tau,\delta \mapsto \textbf{sequence}(e_1,\ldots,e_n) : \textbf{sequence}[T]}$

One element of a sequence can be accessed by indexing:

expression ::= …
| expression**::**expression -- index sequence

[sequence index] $\dfrac{\tau, \delta \mapsto \mathrm{e} : \textbf{sequence}[\,\mathrm{T}\,] \quad \tau, \delta \mapsto \mathrm{e}_1 : \textbf{Integer}}{\tau, \delta \mapsto \mathrm{e} :: \mathrm{e}_1 : \mathrm{T}}$

Mapping to the abstract syntax

The mapping between concrete syntax and abstract syntax is straightforward.

III. σπ-ADL LAYER: TEXTUAL CONCRETE SYNTAX

σπ-ADL is constructed as the outer layer of the ArchWare ADL. It provides style constructs.

It is formally constructed on top of π-ADL and μπ-AAL, and builds the bridge between the two languages, allowing the definition of architectural element styles, represented by property-guarded behaviour abstractions. Its syntax and semantics are presented in deliverable D1.1b. Hereafter, the syntax is summarised.

III.1 Production rules

syntax of behaviours

Behaviour ::= … | StyleInstantiation

StyleInstantiation ::= **style** name{v_1, …, v_n}

syntax of abstractions

BehaviourDefinition ::= **define** compositeName **as** (x_1,…,x_n).Behaviour

| **define** compositeName2 **as** compositeName **{** v_1,…,v_n **}**

-- partial abstraction application

syntax of architectural element styles

ArchitecturalElementStyle ::= **define element style** compositeName **as (**x_1,…,x_n**).**Behaviour **verifying** SetOfProperties

| **define element style** compositeName **as** value**{**v_1,…,v_m**}** **verifying** SetOfProperties

-- sub style by partial abstraction application and additional properties

syntax of properties

property_specification ::= identifier **is property {** property_expression **}**

style_property_specification ::= **hard** property_specification

| **soft** property_specification

SetOfProperties ::= style_property_specification

| style_property_specification SetOfProperties

syntax of types

ElementStyleType ::= $ValueType_1$, …, $ValueType_n$; **Boolean** -> **Behaviour**

III.2 Typing rules

abstraction typing

T-Abstraction1:

$$\frac{\Delta, x_1 : VT_1, \ldots, x_n : VT_n \vdash B : \textbf{Behaviour}}{\Delta \vdash \textbf{define}\ \text{name}\ \textbf{as}\ (x_1,\ldots,x_n).B : VT_1,\ldots,VT_n \rightarrow \textbf{Behaviour}}$$

T-Abstraction2:

$$\frac{\Delta \vdash \text{name} : VT_1, \ldots,\ VT_n \rightarrow \textbf{Behaviour}\ \ \Delta \vdash\ v_{pi1} : VT_{i1}, \ldots, v_{pik} : VT_{ik}}{\Delta \vdash \textbf{define}\ \text{name2}\ \textbf{as}\ \text{name}\{v_{pi1} : VT_{i1}, \ldots, v_{pik} : VT_{ik}\} : VT_{j1},\ldots, VT_{jl} \rightarrow \textbf{Behaviour}}$$

where $\{VT_{i1}, \ldots, VT_{ik}, VT_{j1},\ldots, VT_{jl}\} = \{VT_1,\ldots, VT_n\}, \quad k < n$

T-Application:

$$\frac{\Delta \vdash v : VT_1, \ldots, VT_n \rightarrow \textbf{Behaviour} \qquad \Delta \vdash v_1 : VT_1, \ldots, v_n : VT_n}{\Delta \vdash v\{v_1 : VT_1, \ldots, v_n : VT_n\} : \textbf{Behaviour}}$$

T-ElementStyle:

$$\frac{\Delta, x_1 : VT_1, \ldots, x_n : VT_n \vdash B : \textbf{Behaviour} \qquad \Delta, x_1 : VT_1, \ldots, x_n : VT_n \vdash e : \textbf{Boolean}}{\Delta \vdash \textbf{define element style}\ \text{name}\ \textbf{as}\ (x_1,\ldots,x_n).B\ \ \textbf{verifying}\ e : \text{ValueType}_1, \ldots, \text{ValueType}_n ; \textbf{Boolean} \rightarrow \textbf{Behaviour}}$$

T-StyleInstantiation:

$$\frac{\Delta \vdash\ \text{name} :\ VT_1, \ldots, VT_n ; \textbf{Boolean} \rightarrow \textbf{Behaviour} \qquad \Delta \vdash v_1 : VT_1, \ldots, v_n : VT_n}{\Delta \vdash \textbf{style}\ \text{name}\{v_1 : VT_1, \ldots, v_n : VT_n\} : \textbf{Behaviour}}$$

T-StyleInstantiation:

$$\frac{}{\textbf{style}\ \text{name}\{v_1, \ldots, v_n\} \xrightarrow{\tau} B\ \textbf{where}\ \{v_1, \ldots, v_n\}\ \textbf{rename}\ \{x_1, \ldots, x_n\}}\ condition$$

where condition is

define element style name **as** $(x_1,\ldots, x_n).B$ **verifying** e and $e \equiv \text{true}$

T-SubStyle:

$$\frac{\Delta \vdash v : VT_1, \ldots, VT_n ; \textbf{Boolean} \rightarrow \textbf{Behaviour} \qquad \Delta \vdash\ v_{pi1} : VT_{i1}, \ldots, v_{pik} : VT_{ik} \qquad \Delta, v_{pj1} : VT_{j1}, \ldots, v_{pjl} : VT_{jl} \vdash e : \textbf{Boolean}}{\Delta \vdash \textbf{define element style}\ s\ \textbf{as}\ v\{v_{pi1} : T_{i1}, \ldots, v_{pik} : T_{ik}\}\ \ \textbf{verifying}\ e : VT_{j1}, \ldots, VT_{jl} ; \textbf{Boolean} \rightarrow \textbf{Behaviour}}$$

where $\{VT_{i1}, \ldots, VT_{ik}, VT_{j1},\ldots, VT_{jl}\} = \{VT_1,\ldots, VT_n\}, \quad k < n$

IV. ILLUSTRATING THE USE OF THE ADL

To illustrate the use of the ArchWare ADL for describing software architectures, different architectures are defined hereafter.

A simple architecture of directly composed services

Hereafter a simple architecture for a software that computes the average of a stream of strictly positive real numbers delimited by a zero (the zero denotes the end of the stream). The *counterService* counts the number of numbers of the stream. It terminates when it receives the delimiter (the delimiter is not counted). The *sumService* sums the numbers of the stream. It also terminates when it receives the delimiter (the delimiter is not added). The *averageService* computes the average from results issued by *sumService* and *counterService*, then terminates.

```
value averageArchitecture is abstraction ().{
    value inStream is connection(Real).
    value outResult is connection(Real).
    restrict value outToInNumber is connection(Real).
            value outToInCounter is connection(Real).
            value outToInSum is connection(Real).
    compose {   via counterService send () where {
                        inStream replaces inNumber
                        outToInNumber replaces outNumber
                        outToInCounter replaces outCounter
                }
    and         via sumService send () where {
                        outToInNumber replaces inNumber
                        outToInSum replaces outSum
                }
    and         via averageService send () where {
                        outToInSum replaces inSum
                        outToInCounter replaces inCounter
                        outResult replaces outAverage
                }
    }
} where {
value counterService is abstraction ().{
    value inNumber is connection(Real).
    value outNumber is connection(Real).
    value outCounter is connection(Real).
```

```
        restrict value counter is location(0).
        via inNumber  receive x : Real.
        via outNumber send x.
        if (x ~= 0)        then    {via counter locate 'counter + 1.
                                   via counterService send ()}
                           else    {via outCounter send 'counter.
                                   done}
}
value sumService is abstraction ().{
        value inNumber is connection(Real).
        value outSum is connection(Real).
        restrict value sum is location(0).
        via inNumber  receive x : Real.
        if (x ~= 0)        then    {via sum locate 'sum + 1.
                                   via sumService send ()}
                           else    {via outSum send 'sum.
                                   done}
}
value averageService is abstraction ().{
        value inSum is connection(Real).
        value inCounter is connection(Real).
        value outAverage is connection(Real).
        via inSum receive sum : Real.
        via inCounter receive counter : Real.
        if counter ~= 0    then    via outAverage send sum / counter
                           else    via outAverage send 0.
        done
}
} -- end of averageArchitecture --
```

The *averageArchitecture* is itself a service, in fact it is a composite service that can be used in other architectures. For instance, it can be used in another architecture that computes the average of numbers in a given range. The *generatorService* outputs the numbers from *lowerBound* to *upperBound*, followed by a *eos* (end-of-stream).

```
value generatorService is abstraction (lowerBound, upperBound, eos : Integer).{
        value outGenerator is connection(Integer).
        iterate sequence(lowerBound .. upperBound) by x : Integer
                do via outGenerator send x.
        via outGenerator send eos.
        done
```

```
}
```

The architecture composed of the *generatorService* connected to the *averageArchitecture* connected to a print behaviour is described as follows.

```
value rangeAverageArchitecture is abstraction (lb, ub, eos : Integer).{
    restrict value outGeneratorToInStream is connection(Integer).
            value outResultToInPrint is connection(Real).
    compose {   via generatorService send (lb, ub, eos) where {
                        outGeneratorToInStream replaces outGenerator
                }
    and         via averageArchitecture send () where {
                        outGeneratorToInStream replaces inStream
                        outResultToInPrint replaces outResult
                }
    and         behaviour {via outResultToInPrint receive result : Real.
                        via writeString send "The average of the range is ".
                        via writeReal send result}
    }
```

In order to run the described architecture, one can evaluate:

```
behaviour {via rangeAverageArchitecture send (2, 10, 0)}
```

A simple architecture of piped composed services

Let us redefine the *averageArchitecture* in order to introduce pipes between services. A *pipeService* buffers numbers between services in the architecture.

```
value averagePipedArchitecture is abstraction ().{
    name inStream : connection[Real].
    name outResult : connection[Real].
    restrict name outNumber_inPipeCounterSum : connection[Real].
            name outPipeCounterSum_inNumber : connection[Real].
            name outCounter_inPipeCounterAverage : connection[Real].
            name outPipeCounterAverage_inCounter : connection[Real].
            name outSum_inPipeSumAverage : connection[Real].
            name outPipeSumAverage_inSum : connection[Real].
            value pipeCounterSumService is pipeService.
            value pipeCounterAverageService is pipeService.
            value pipeSumAverageService is pipeService.
    compose {   via counterService send () where {
                        inStream renames inNumber.
                        outNumber_inPipeCounterSum renames outNumber.
```

```
                    outCounter_inPipeCounterAverage renames
outCounter
                }
    and         via pipeCounterSumService send () where {
                    outNumber_inPipeCounterSum renames inPipe.
                    outPipeCounterSum_inNumber renames outPipe
                }
    and         via pipeCounterAverageService send () where {
                    outCounter_inPipeCounterAverage renames inPipe
                    outPipeCounterAverage_inCounter renames outPipe
                }
    and         via sumService send () where {
                    outPipeCounterSum_inNumber renames inNumber.
                    outSum_inPipeSumAverage renames outSum
                }
    and         via pipeSumAverageService send () where {
                    outSum_inPipeSumAverage renames inPipe
                    outPipeSumAverage_inSum renames outPipe
                }
    and         via averageService send () where {
                    outPipeSumAverage_inSum renames inSum.
                    outPipeCounterAverage_inCounter renames inCounter.
                    outResult renames outAverage
                }
    }
} where {
value counterService is abstraction ().{
    name inNumber : connection[Real].
    name outNumber : connection[Real].
    name outCounter : connection[Real].
    restrict value counter is location(0).
    via inNumber  receive x : Real.
    via outNumber send x.
    if (x ~= 0)     then    {via counter locate 'counter + 1.
                            via counterService send ()}
                    else    {via outCounter send 'counter.
                            done}
}.
value sumService is abstraction ().{
```

```
        name inNumber : connection[Real].
        name outSum : connection[Real].
        restrict value sum is location(0).
        via inNumber  receive x : Real.
        if (x ~= 0)        then    {via sum locate 'sum + 1.
                                   via sumService send ()}
                           else    {via outSum send 'sum.
                                   done}
    }.
    value averageService is abstraction ().{
        name inSum : connection[Real].
        name inCounter : connection[Real].
        name outAverage : connection[Real].
        via inSum receive sum : Real.
        via inCounter receive counter : Real.
        if counter ~= 0   then    via outAverage send sum / counter
                          else    via outAverage send 0.
        done
    }.
    value pipeService is abstraction (eos : Real).{
        name inPipe : connection[Real].
        name outPipe : connection[Real].
        replicate via inPipe receive x : Real.
        via outPipe send x.
        done
    }
    } -- end of averagePipedArchitecture --
```

The *averagePipedArchitecture* can be used in the *rangeAverageArchitecture* described so far. The architecture composed of the *generatorService* connected to the *averagePipedArchitecture* connected to a print behaviour can be described as follows.

```
    value rangeAverageArchitecture is abstraction (lb, ub, eos : Integer).{
        restrict value outGeneratorToInStream is connection(Integer).
                value outResultToInPrint is connection(Real).
        compose {   via generatorService send (lb, ub, eos) where {
                                outGeneratorToInStream replaces outGenerator
                    }
        and         via averagePipedArchitecture send () where {
                                outGeneratorToInStream replaces inStream
                                outResultToInPrint replaces outResult
```

```
                    }
    and             behaviour {via outResultToInPrint receive result : Real.
                            via writeString send "The average of the range is ".
                            via writeReal send result}
    }
}
```

In order to run the described architecture, one can evaluate:

```
behaviour {via rangeAverageArchitecture send (2, 10, 0)}
```

A simple architecture of piped connected services

Now let us redefine the *averagePipedArchitecture* in order to introduce another expression for connecting connections.

```
value averageConnectedArchitecture is abstraction ().{
    compose {   via counterService send ()
    and         via sumService send ()
    and         via averageService send ()
    and         {via counterServiceMetaPort receive counterServicePort : view[
                            inNumber : connection[Real],
                            outNumber : connection[Real],
                            outCounter : connection[Real]].
                via sumServiceMetaPort receive sumServicePort : view[
                            inNumber : connection[Real],
                            outSum : connection[Real]].
                via averageServiceMetaPort receive averageServicePort : view[
                            inSum : connection[Real],
                            inCounter: connection[Real],
                            outAverage : connection[Real]].
    and         via connectionService send
                        (counterServicePort::outNumber,
                        sumServicePort::inNumber)
                via connectionService send
                        (counterServicePort::outCounter,
                        averageServicePort::inCounter)
                via connectionService send
                        (sumServicePort::outSum, averageServicePort::inSum)
    and         via connectionService send
    } where {   inStream renames counterServicePort::inNumber.
                outResult renames averageServicePort::outAverage.
```

```
} where {
value counterService is abstraction ().{
    name counterServiceMetaPort : connection[
        view[   inNumber : connection[Real],
                outNumber : connection[Real],
                outCounter : connection[Real]]].
    value port is
        view(   inNumber is connection(Real),
                outNumber is connection(Real),
                outCounter is connection(Real))
    restrict value counter is location(0).
    via counterServiceMetaPort send port.
    via port::inNumber  receive x : Real.
    via port::outNumber send x.
    if (x ~= 0)     then    {via counter locate 'counter + 1.
                            via counterService send ()}
                    else    {via port::outCounter send 'counter.
                            done}
}.
value sumService is abstraction ().{
    name sumServiceMetaPort : connection[
        view[   inNumber : connection[Real],
                outSum : connection[Real]]].
    value port is
        view(   inNumber is connection(Real),
                outSum is connection(Real))
    restrict value sum is location(0).
    via sumServiceMetaPort send port.
    via port::inNumber  receive x : Real.
    if (x ~= 0)     then    {via sum locate 'sum + 1.
                            via sumService send ()}
                    else    {via port::outSum send 'sum.
                            done}
}.
value averageService is abstraction ().{
    name averageServiceMetaPort : connection[
        view[   inSum : connection[Real],
                inCounter : connection[Real],
```

```
                    outAverage : connection[Real]]].
    value port is
            view(   inNumber is connection(Real),
                    outNumber is connection(Real),
                    outCounter is connection(Real))
    via averageServiceMetaPort send port.
    via port::inSum receive sum : Real.
    via port::inCounter receive counter : Real.
    if counter ~= 0  then   via port::outAverage send sum / counter
                     else   via port::outAverage send 0.
    done
}.
value connectionService is abstraction
            (inConnection : connection[connection[Real]],
            outConnection : connection[connection[Real]]).{
    via inConnection receive fromConnection : connection[Real].
    via outConnection receive toConnection : connection[Real].
    replicate via fromConnection receive x : Real.
    via toConnection send x.
    done
}
} -- end of averageConnectedArchitecture --
```

Appendix A : π-ADL production rules

Declarations

script ::= declaration [**;** script] | clause [**;** script]

declaration ::= value_declaration | type_declaration

Type declaration

type_declaration ::= **let type** type_definition |
recursive let type type_definition [**&** type_definition]*

type_definition ::= identifier **be** type

Value declaration

value_declaration ::= **let** identifier_clause_list |
recursive let identifier_literal_list

identifier_clause_list ::= identifier = clause[, identifier_clause_list]

identifier_literal_list ::= identifier = literal [**&** identifier_literal_list]

Type descriptors

type ::= **Natural | Integer | Real | Boolean | String | Any |**
connection [type **] | behaviour | behaviour [**type_list**]** |
tuple [type_list **]** | **view [** identifier_type_list **] | union [** type_list **]** |
variant [identifier_type_list **]**
| **location [** type **] | sequence [** type **]** |
set [type **]** | **bag [** type **] | quote** identifier

type_list ::= type [**,** type_list]

identifier_type_list ::= identifier **:** type [, identifier_type_list]

identifier_type_list_2 ::= identifier identifier_type_list_2
| identifier **:** type identifier_type_list_2

Clauses (includes behaviours)

clause ::= **restrict** identifier_type_list **in {**clause**}**-- restriction
| prefix **then** clause -- prefix
| **choose [** choice_list **]** -- choice
| **compose [** paralel_list **]** -- composition

```
                |   replicate clause                          -- replication
                |   if clause do clause                -- match prefix
                |   project clause as type onto project_list default : clause
                                                              -- projection from any, union etc.
                |   project clause as identifier onto project_list default : clause
                                                              ! projection from variant
                |   name := clause
                |   expression
choice_list     ::= clause [or choice_list]
paralel_list    ::= clause [and choice_list]
prefix          ::= via expression send clause                -- output prefix
                |   via expression receive identifier_type_list_2  -- input prefix
                |   unobservable                              -- silent prefix
identifier_list     ::= identifier [, identifier]
```

Expressions

```
expression      ::=     ( clause ) |
                        { script } |                          -- new scope
                        literal |                             -- literal values
                        not expression |                      -- boolean expressions
                        expression and expression  |
                        expression or expression |
                        expression implies expression |
                        add_operator expression |                   -- signed expression
                        expression relational_operator expression |  -- comparison expression
                        expression  add_operator expression |       -- arithmetic expression
                        expression  multiply_operator expression |
                        expression ++ expression |            -- string and sequence concatenation
                        expression ⟪ clause ‖ clause ⟫ |             -- substring
                        expression collection_operator expression | -- collection expressions
                        any⟪ clause ⟫ |                             -- injection into any
                        rename clause using ⟦renaming_list⟧ |       -- substitution
                        name [name_list] :: clause |          -- name channels in a behaviour
                        tuple ( clause_list ) |                   -- make tuple
                        use identifier_list from identifier |     -- project from tuple
                        view ( identifier_clause_list ) |         -- make view
                        clause . identifier |                     -- dereference view
                        union ( clause ) : type |                 -- make union
```

	variant (identifier **=** clause **) :** type |	-- make variant
	location (clause **)** |	-- make location
	' clause |	-- dereference location
	sequence (clause_list **)** |	-- make sequence
	clause**[** clause **]** |	-- index sequence
	set (clause_list **)** |	-- make set
	bag (clause_list **)** |	-- make bag
	one of clause |	-- select from set/bag
	rest of clause |	-- select from set/bag
	intersection(expression**,**expression**)** |	-- intersect set/bag
	reunion(expression**,**expression**)** |	-- reunion set/bag
	name |	
	identifier**(**application_list**)**	-- abstraction application

renaming_list	::=	name **as** name [**,** renaming_list]
clause_list	::=	clause [, clause_list]
name_list	::=	name [, name_list]
name	::=	identifier
relational_operator	::=	equality_operator | ordering_operator
equality_operator	::=	= | <>
ordering_operator	::=	< | <= | > | =>
add_operator	::=	+ | -
multiply_operator	::=	integer_multiply_operator | real_multiply_operator
integer_multiply_operator	::=	* | **div** | **rem**
real_multiply_operator	::=	* | /
collection_operator	::=	∪ | ∩

Literals

literal	::=	natural_literal | integer_literal | real_literal | boolean_literal | string_literal |connection_literal | behaviour_literal | set_literal | bag_literal | abstraction_literal

natural_literal	::=	#digit [digit]*
integer_literal	::=	[add_operator] digit [digit]*
real_literal	::=	integer_literal.[digit]*[**e** integer_literal]
boolean_literal	::=	**true** | **false**
string_literal	::=	"[character]*"
connection_literal	::=	**connection** ⟪ type ⟫
set_literal	::=	**set()**

bag_literal ::= **bag()**

behaviour_literal ::= **done**

abstraction_literal ::= **abstraction (**identifier_type_list**);** clause

application_list ::= clause **as** identifier [**;** application_list]

Names and labels

identifier ::= letter [letter | digit | _]*

letter ::= **a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z**

digit ::= **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

character ::= any ASCII character

Appendix B: π-ADL typing rules

The type rules of π-ArchWare[PB] ADL are used in conjunction with the context free syntax to determine the legal set of (type correct) programs. For this a method of specifying the type rules is required.

Before that, however, the concept of environments is introduced. Two kinds of environments are used, both of which are sets of bindings: for value identifiers and for type identifiers. δ denotes the environments where value identifiers are bound to their types in the form of <x, T>, where x is an identifier and T is a type. τ stands for the environment in which type identifiers are bound to type expressions in the form <t, T>, where t is an identifier and T is a type. $A_1::b::A_2$ is used to represent a list A which contains a binding b. A++B is used to denote the concatenation of two lists of bindings A and B. δ_V, δ_A, τ_V and τ_A are global environments and support block structure.

As introduced above bindings are represented as pairs and the notation < x, T > is used to denote a pair value consisting of x and T. (b_1, …, b_n) is a list containing bindings b_1 to b_n. The meta function *type_declaration* takes a list of bindings between value type identifiers and value type expressions and adds them to the environment τ. Similarly, the meta function *id_declaration* takes a list of bindings between identifiers and types as its arguments and updates the environment δ with the new bindings.

The typing rules use the structure of proof rules i.e. if A_i is true for i = 1, . . n then B is true. Each A_i and B may be of the form $X \mapsto Y$ which, in the context of this document, is used to denote that Y is deducible from a collection of environments X. Thus, the type rule

$$\frac{\tau, \delta \mapsto e_1 : \textbf{Integer} \quad \tau\ \delta \mapsto e_2 : \textbf{Integer}}{\tau, \delta \mapsto e_1 + e_2 : \textbf{Integer}}$$

is read as "if expression e_1 is deduced to be of type **Integer** from environments τ and δ and expression e_2 is deduced to be of type **Integer** from environments τ and δ then the type of the expression e_1 + e_2 can be deduced to be of type **Integer** from environments τ and δ.

Declarations

[declarations]

$$\frac{\tau, \delta \mapsto D_1 : \textbf{void} \quad \text{type\_declaration}(\Omega_1) \quad \text{id\_declaration}(\Psi_1) \quad \tau{+}{+}\Omega_1, \delta{+}{+}\Psi_1 \mapsto D_2 : T}{\tau, \delta \mapsto D_1;D_2 : T}$$

[type declaration]

$$\frac{\tau \mapsto T \in \text{ValueType}}{\tau, \delta \mapsto \textbf{let type}\ \text{name}\ \textbf{be}\ T : \textbf{void} \quad \text{type\_declaration}(<\text{name}, T>)}$$

[recursive type declaration]

$$\frac{\tau \mapsto T_1 \in \text{Type} \quad \tau \mapsto T_2 \in \text{Type}}{\tau' \mapsto \textbf{recursive let type}\ t_1\ \textbf{be}\ T_1\ \textbf{\&}\ t_2\ \textbf{be}\ T_2 : \textbf{void} \quad \text{type\_declaration}(<t_1, T_1>, <t_2, T_2>)}$$

where τ stands for $\tau_1::\tau_2::\tau_3$ and τ' stands for $\tau_1::<t_1, T_1>::\tau_2::<t_2, T_2>::\tau_3$

[value declaration] $$\frac{\tau \mapsto T \in \text{ ValueType} \quad \tau, \delta \mapsto e : T}{\tau, \delta \mapsto \textbf{let } \text{name} = e : \textbf{void} \quad \text{id\_declaration}(<\text{name}, T>)}$$

[recursive value declaration]

$$\frac{\tau, \delta \mapsto e_1 : T_1 \quad \tau, \delta \mapsto e_2 : T_2}{\tau, \delta' \mapsto \textbf{recursive let } i_1 = e_1 \,\&\, i_2 = e_2 : \textbf{void} \quad \text{id\_declaration}(<i_1, T_1>, <i_2, T_2>)}$$ where

δ stands for $\delta_1 :: \delta_2 :: \delta_3$ and δ' stands for $\delta_1 :: <i_1, T_1> :: \delta_2 :: <i_2, T_2> :: \delta_3$

Clauses

[restriction]

$$\frac{\tau \mapsto T_1 \in \text{Connection Type} \ \ldots \tau \mapsto T_n \in \text{Connection Type} \quad \tau, \delta :: <i_1, T_1> :: \ldots :: <i_n, T_n> \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{restrict } i_1 : T_1, \ldots, i_n : T_n \textbf{ in } \{B\} : \textbf{behaviour} \quad \text{id\_declaration}(<i_1, T_1>, \ldots, <i_n, T_n>)}$$

[composition] $$\frac{\tau, \delta \mapsto B_1 : \textbf{behaviour} \quad \tau, \delta \mapsto B_2 : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{compose } [B_1 \textbf{ and } B_2] : \textbf{behaviour}}$$

[choice] $$\frac{\tau, \delta \mapsto B_1 : \textbf{behaviour} \quad \tau, \delta \mapsto B_2 : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{choose } [B_1 \textbf{ or } B_2] : \textbf{behaviour}}$$

[conditioned behaviour] $$\frac{\tau, \delta \mapsto \text{expr} : \textbf{Boolean} \quad \tau, \delta \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{if } \text{expr } \textbf{do } B : \textbf{behaviour}}$$

[replication] $$\frac{\tau, \delta \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{replicate } B : \textbf{behaviour}}$$

[unobservable] $$\frac{}{\tau, \delta \mapsto \textbf{unobservable} : \textbf{void}}$$

[receive 1] $$\frac{\tau \mapsto T \in \text{ValueType } \tau, \delta \mapsto \text{name} : \textbf{connection}\{T\} \quad \tau, \delta \mapsto x : T}{\tau, \delta \mapsto \textbf{via } \text{name } \textbf{receive } x : \textbf{void}}$$

[receive 2] $$\frac{\tau \mapsto T \in \text{ValueType } \tau, \delta \mapsto \text{name} : \textbf{connection}\{T\}}{\tau, \delta \mapsto \textbf{via } \text{name } \textbf{receive } x : T : \textbf{void} \quad \text{id\_declaration}(<x, T>)}$$

[send] $$\frac{\tau \mapsto T \in \text{ValueType} \quad \tau, \delta \mapsto \text{name} : \textbf{connection}\{T\} \quad \tau, \delta \mapsto x : T}{\tau, \delta \mapsto \textbf{via } \text{name } \textbf{send } x : \textbf{void}}$$

Expressions

Boolean

[negation] $$\frac{\tau, \delta \mapsto e : \textbf{Boolean}}{\tau, \delta \mapsto \textbf{not } e : \textbf{Boolean}}$$

[or] $$\frac{\tau, \delta \mapsto e_1 : \textbf{Boolean} \quad \tau, \delta \mapsto e_2 : \textbf{Boolean}}{\tau, \delta \mapsto e_1 \textbf{ or } e_2 : \textbf{Boolean}}$$

[and] $$\frac{\tau, \delta \mapsto e_1 : \textbf{Boolean} \quad \tau, \delta \mapsto e_2 : \textbf{Boolean}}{\tau, \delta \mapsto e_1 \textbf{ and } e_2 : \textbf{Boolean}}$$

[implies] $$\frac{\tau, \delta \mapsto e_1 : \textbf{Boolean} \quad \tau, \delta \mapsto e_2 : \textbf{Boolean}}{\tau, \delta \mapsto e_1 \textbf{ implies } e_2 : \textbf{Boolean}}$$

Comparison

[equality] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 = e_2 : \textbf{Boolean}}$$

[non equality] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 \sim= e_2 : \textbf{Boolean}}$$

In the following, T ∈ { **Natural**, **Integer**, **Real**, **String** }

[less] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 < e_2 : \textbf{Boolean}}$$

[less equal] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 <= e_2 : \textbf{Boolean}}$$

[greater] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 > e_2 : \textbf{Boolean}}$$

[greater equal] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 >= e_2 : \textbf{Boolean}}$$

Numeric Expression

In the following type rules, T ∈ { **Integer**, **Real** }, T' ∈ { **Natural**, **Integer**, **Real** } and T" ∈ { **Natural**, **Integer** }.

[plus] $$\frac{\tau,\delta \mapsto e : T}{\tau,\delta \mapsto + e : T}$$

[minus] $$\frac{\tau,\delta \mapsto e : T}{\tau,\delta \mapsto - e : T}$$

[add] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 + e_2 : T'}$$

[subtract] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 - e_2 : T'}$$

[times] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 * e_2 : T'}$$

[division] $$\frac{\tau,\delta \mapsto e_1 : T'' \;\; \tau,\delta \mapsto e_2 : T''}{\tau,\delta \mapsto e_1 \textbf{ div } e_2 : T''}$$

[remainder] $$\frac{\tau,\delta \mapsto e_1 : T'' \;\; \tau,\delta \mapsto e_2 : T''}{\tau,\delta \mapsto e_1 \textbf{ rem } e_2 : T''}$$

[real division] $$\frac{\tau,\delta \mapsto e_1 : \textbf{Real} \;\; \tau,\delta \mapsto e_2 : \textbf{Real}}{\tau,\delta \mapsto e_1 / e_2 : \textbf{Real}}$$

String Expression

[concatenation] $$\frac{\tau,\delta \mapsto e_1 : \textbf{String} \;\; \tau,\delta \mapsto e_2 : \textbf{String}}{\tau,\delta \mapsto e_1 ++ e_2 : \textbf{String}}$$

[substring] $$\frac{\tau,\delta \mapsto e : \textbf{String} \quad \tau,\delta \mapsto e_1 : \textbf{Natural} \quad \tau,\delta \mapsto e_2 : \textbf{Natural}}{\tau,\delta \mapsto e\,(\,e_1 \,|\, e_2\,) : \textbf{String}}$$

Collection Expression

[included equal] $\dfrac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 <= e_2 : \textbf{Boolean}}$ where T ∈ { **set**, **bag** }

[included] $\dfrac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 < e_2 : \textbf{Boolean}}$ where T ∈ { **set**, **bag** }

[includes equal] $\dfrac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 >= e_2 : \textbf{Boolean}}$ where T ∈ { **set**, **bag** }

[includes] $\dfrac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 > e_2 : \textbf{Boolean}}$ where T ∈ { **set**, **bag** }

[intersection] $\dfrac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto \textbf{intersection}(e_1, e_2) : T}$ where T ∈ { **set**, **bag** }

[reunion] $\dfrac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto \textbf{reunion}(e_1, e_2) : T}$ where T ∈ { **set**, **bag** }

[concatenation] $\dfrac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 ++ e_2 : T}$ where T ∈ { **sequence** }

Literals

[natural literal] $\dfrac{}{\tau,\delta \mapsto n : \textbf{Natural}}$ n ∈ Natural

[integer literal] $\dfrac{}{\tau,\delta \mapsto i : \textbf{Integer}}$ i ∈ Integer

[real literal] $\dfrac{}{\tau,\delta \mapsto r : \textbf{Real}}$ r ∈ Real

[boolean literal] $\dfrac{}{\tau,\delta \mapsto b : \textbf{Boolean}}$ b ∈ Boolean

[string literal] $\dfrac{}{\tau,\delta \mapsto s : \textbf{String}}$ s ∈ String

[connection literal] $\dfrac{\tau \mapsto T \in \text{ValueType}}{\tau,\delta \mapsto \textbf{connection(}T\textbf{)} : \textbf{connection[}T\textbf{]}}$

[behaviour literal] $\dfrac{}{\tau,\delta \mapsto \textbf{done} : \textbf{behaviour}}$

[quote literal] $\dfrac{i \in \text{Identifier}}{\tau \mapsto i : \textbf{quote}\ \ i}$

[abstraction literal]

$$\frac{\tau \mapsto VT_1 \in \text{ValueType}, \ldots, \tau \mapsto VT_n \in \text{ValueType} \quad \tau,\delta ::< x_1, VT_1 >:: \ldots ::< x_n, VT_n > \mapsto B : \textbf{behaviour}}{\tau,\delta \mapsto \textbf{abstraction}(x_1 : VT_1, \ldots, x_n : VT_n); B : \ \textbf{behaviour[}VT_1, \ldots, VT_n\textbf{]}}$$

Block

[parenthesis] $\dfrac{\tau,\delta \mapsto e : T}{\tau,\delta \mapsto \textbf{(}e\textbf{)} : T}$

[{}] $\dfrac{\tau, \delta \mapsto e : T}{\tau, \delta \mapsto \{e\} : T}$

Abstraction

[abstraction derivation]

$$\frac{\text{Decl1} \quad \tau, \delta \mapsto A : \textbf{behaviour}[VT_1, \ldots, VT_n] \; \tau, \delta \mapsto x_{pi1} : VT_{i1}, \ldots, \tau, \delta \mapsto x_{pik} : VT_{ik}}{\tau, \delta \mapsto A(x_{pi1} \textbf{ as } x_{i1}, \ldots, x_{pik} \textbf{ as } x_{ik}) : \textbf{behaviour}[VT_{j1}, \ldots, VT_{jl}] \quad \text{Decl2}} \text{ Where}$$

Decl1 stands for $\tau \mapsto VT_1 \in \text{ValueType}, \ldots, \tau \mapsto VT_n \in \text{ValueType}$

Decl2 stands for

$\text{id\_declaration}(< x_{pi1} : VT_{i1} >, \ldots, < x_{pik} : VT_{ik} >)$ and $\{VT_{i1}, \ldots, VT_{ik}, VT_{j1}, \ldots, VT_{jl}\} = \{VT_1, \ldots, VT_n\}, k < n$

Behaviour

[abstraction application]

$$\frac{\text{Decl1} \quad \tau, \delta \mapsto A : \textbf{behaviour}[VT_1, \ldots, VT_n] \; \tau, \delta \mapsto x_{pi1} : VT_{i1}, \ldots, \tau, \delta \mapsto x_{pin} : VT_{in}}{\tau, \delta \mapsto A(x_{pi1} \textbf{ as } x_{i1}, \ldots, x_{pin} \textbf{ as } x_{in}) : \textbf{behaviour} \quad \text{Decl2}} \text{ where}$$

Decl1 stands for $\tau \mapsto VT_1 \in \text{ValueType}, \ldots, \tau \mapsto VT_n \in \text{ValueType}$

Decl2 stands for $\text{id\_declaration}(< x_{pi1}, VT_{i1} >, \ldots, < x_{pik}, VT_i >)$

Identifier

[identifier] $\dfrac{}{\tau, \delta_1 :: < x, T > :: \delta_2 \mapsto x : T}$

[renaming] $\dfrac{\tau \mapsto T1, \ldots, Tn \in \text{ConnectionType} \quad \tau, \delta \mapsto e : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{rename } e \textbf{ using [} x \textbf{ as } y \textbf{]} : \textbf{void} \;\; \text{id\_substitution}(< x, y >)}$

[naming]

$$\frac{\tau, \delta \mapsto e : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{naming } [x_1, \ldots, x_n] :: e : \textbf{void} \;\; \text{id\_declaration}(< x_1, \textbf{behaviour} >, \ldots, < x_n, \textbf{behaviour} >)}$$

Tuple

[tuple value] $\dfrac{\tau \mapsto T1, \ldots, Tn \in \text{ValueType} \quad \tau, \delta \mapsto e_1 : T_1 \ldots \tau, \delta \mapsto e_n : T_n}{\tau, \delta \mapsto \textbf{tuple}(e_1, \ldots, e_n) : \textbf{tuple}[T_1, \ldots, T_n]}$

[tuple dereference]

$$\frac{\tau \mapsto T1, \ldots, Tn \in \text{ValueType} \quad \tau, \delta \mapsto e : \textbf{tuple}[T_1, \ldots, T_n]}{\tau, \delta \mapsto \textbf{use } x_1, \ldots, x_n \textbf{ from } e : \textbf{void} \;\; \text{id\_declaration}(< x_1, T_1 >, \ldots, < x_n, T_n >)}$$

View

[view value] $\dfrac{\tau \mapsto T1, \ldots, Tn \in \text{ValueType} \quad \tau, \delta \mapsto v_1 : T_1 \;\ldots\; \tau, \delta \mapsto v_n : T_n}{\tau, \delta \mapsto \textbf{ view } (l_1 = v_1, \ldots, l_n = v_n) : \textbf{view}[l_1 : T_1, \ldots, l_n : T_n]}$

[view dereference] $\dfrac{\tau \mapsto T1, \ldots, Tn \in \text{ValueType} \quad \tau, \delta \mapsto v : \textbf{view}[l_1 : T_1, \ldots, l_n : T_n]}{\tau, \delta \mapsto \; v.l_i : T_i}$

Union

[union-i value] $$\frac{\tau, \delta \mapsto v : T_i}{\tau_1 ::< T, \textbf{union[}T_1 \ldots T_n\textbf{]} >:: \tau_2, \delta \mapsto \textbf{union}(v) : T : \textbf{union[}T_1 \ldots T_n\textbf{]}}$$

[union projection]

$$\frac{\tau, \delta \mapsto e : \textbf{union[}T_1, \ldots, T_n\textbf{]} \quad \forall i \in \{1,\ldots,n+1\}(\tau, \delta_1 ::< x, T_i >:: \delta_2 \mapsto e_i : \textbf{T)}}{\tau, \delta \mapsto \textbf{project}\ e\ \textbf{as}\ x\ \textbf{onto}\ \{T_1 : e_1; \ldots, T_n : e_n ; \textbf{default}\ e_{n+1}\} : T \quad \text{id\_declaration}(< x, T_i >)}$$

Infinite Union

[any-injection] $$\frac{\tau \mapsto T \in \text{ValueType} \qquad \tau, \delta \mapsto v : T}{\tau, \delta \mapsto \textbf{any}(v) : \textbf{Any}}$$

[any projection]

$$\frac{\tau \mapsto T1, \ldots, Tn \in \text{ValueType} \quad \tau, \delta \mapsto v : \textbf{Any} \quad \forall i \in \{1,\ldots,n+1\}(\tau, \delta_1 ::< x, T_i >:: \delta_2 \mapsto e_i : \textbf{T)}}{\tau, \delta \mapsto \textbf{project}\ e\ \textbf{as}\ x\ \textbf{onto}\ \{T_1 : e_1; \ldots, T_n : e_n ; \textbf{default}\ e_{n+1}\} : T \quad \text{id\_declaration}(< x, T_i >)}$$

Variant

[variant value]

$$\frac{\tau \mapsto T1, \ldots, Tn \in \text{ValueType} \qquad \tau, \delta \mapsto v : T_i}{\tau_1 ::< T, \textbf{variant[}l_1 : T_1, \ldots, l_i : T_i, \ldots, l_n : T_n\textbf{]} >:: \tau_2, \delta \mapsto \textbf{variant}(l_i = v) : T : \textbf{variant[}l_1 : T_1, \ldots, l_i : T_i, \ldots, l_n : T_n\textbf{]}}$$

[variant projection]

$$\frac{\tau, \delta \mapsto e : \textbf{variant[}l_1 : T_1, \ldots, l_i : T_i, \ldots, l_n : T_n\textbf{]} \quad \forall i \in \{1,\ldots,n+1\}(\tau, \delta_1 ::< x, T_i >:: \delta_2 \mapsto e_i : \textbf{T)}}{\tau, \delta \mapsto \textbf{project}\ e\ \textbf{as}\ x\ \textbf{onto}\ \{l_1 : e_1; \ldots, l_n : e_n ; \textbf{default}\ e_{n+1}\} : T \quad \text{id\_declaration}(< x, T_i >)}$$

Location

[location value] $$\frac{\tau \mapsto T \in \text{ValueType} \qquad \tau, \delta \mapsto v : T}{\tau, \delta \mapsto \textbf{location}\ (v) : \textbf{location[}T\textbf{]}}$$

[location dereference] $$\frac{\tau \mapsto T \in \text{ValueType} \qquad \tau, \delta \mapsto l : \textbf{location[}T\textbf{]}}{\tau, \delta \mapsto\ 'l : T}$$

[assignment] $$\frac{\tau \mapsto T \in \text{ValueType} \quad \tau, \delta \mapsto v : T \quad \tau, \delta \mapsto n : \textbf{location[}T\textbf{]}}{\tau, \delta \mapsto v := n : \textbf{void}}$$

Sequence

[sequence value] $$\frac{\tau, \delta \mapsto e_1 : T \ldots \tau, \delta \mapsto e_n : T}{\tau, \delta \mapsto \textbf{sequence}(e_1, \ldots, e_n) : \textbf{sequence}[T]}$$

[sequence index] $$\frac{\tau, \delta \mapsto e : \textbf{sequence}[T] \quad \tau, \delta \mapsto e_1 : \textbf{Integer}}{\tau, \delta \mapsto e[e_1] : T}$$

Set

[set value] $$\frac{\tau, \delta \mapsto e_1 : T \ldots \tau, \delta \mapsto e_n : T}{\tau, \delta \mapsto \textbf{set}(e_1, \ldots, e_n) : \textbf{set}[T]}$$

[set selection - 1]

$$\frac{\tau, \delta \mapsto e : \textbf{set}[\,T\,]}{\tau, \delta \mapsto \textbf{one of}\ e : T}$$

[set selection - 2]

$$\frac{\tau, \delta \mapsto e : \textbf{set}[\,T\,]}{\tau, \delta \mapsto \textbf{rest of}\ e :\ \textbf{set}[\,T\,]}$$

Bag

[bag value]

$$\frac{\tau, \delta \mapsto e_1 : T \ldots \tau, \delta \mapsto e_n : T}{\tau, \delta \mapsto \textbf{bag}(\,e_1, \ldots, e_n) : \textbf{bag}[\,T\,]}$$

[bag selection - 1]

$$\frac{\tau, \delta \mapsto e : \textbf{bag}[\,T\,]}{\tau, \delta \mapsto \textbf{one of}\ e : T}$$

[bag selection - 2]

$$\frac{\tau, \delta \mapsto e : \textbf{bag}[\,T\,]}{\tau, \delta \mapsto \textbf{rest of}\ e :\ \textbf{bag}[\,T\,]}$$